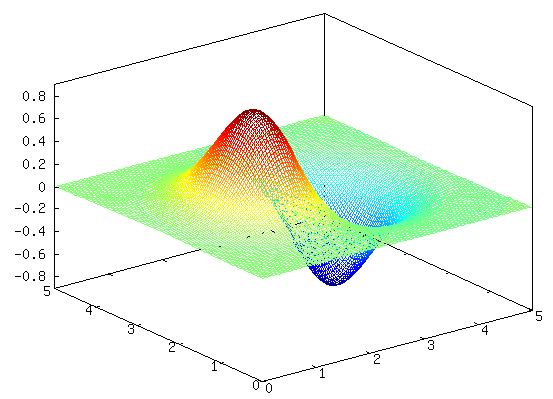
Università di Roma Tor Vergata - Facoltà di Ingegneria  
Corso di laurea magistrale Ingegneria Informatica

Performance Modeling of Computer System and Networks

a.a 2014/2015

Empirical test of randomness and web-server simulation



componenti del gruppo di lavoro:

Marco Bonuglia 0211386 marco.bonuglia85@gmail.com  
Marco Rosati 0211656 marcorosati1988@gmail.com  
Honorine Nakure 0210335 Honorinenakure@yahoo.fr  
  
ultima versione:  
Settembre 2015

**INDICE**

[PARTE 1](#h.6syvhyrfpbn1)

[Introduzione](#h.2fqhgitlykww)

[Capitolo 1  
Implementazione del generatore Lehmer](#h.sk4q2ap0ftyp)

[Scelta del moltiplicatore](#h.hvq869i7w4lb)

[Test spettrale](#h.iltvwb7ek89h)

[Calcolo del Jump Multiplier](#h.exod3pok9li4)

[Capitolo 2](#h.rv0pi7hfwnhl)

[Extremes Test](#h.ax45c45k62x1)

[Confronto con moltiplicatore 48271 e 16807](#h.8ajlavt3yoqo)

[Analisi di Kolmogorov-Smirnov](#h.gtnzjsij91pv)

[Conclusioni](#h.hgzlp9y31qqp)

[APPENDICE A](#h.k9e3gshektxx)

[PARTE 2  
Simulatore di un sistema web multi-tier](#h.9cqrt5el8w41)

[Introduzione](#h.43lg7kspse88)

[Capitolo 1](#h.gqnn5rrhaj99)

[Obiettivi](#h.q9hmwg7s3euo)

[Capitolo 2](#h.84ce4mdqw25u)

[Modello concettuale](#h.rx4igkrbrzmc)

[Variabili di stato](#h.agi9f8pr6fzl)

[Capitolo 3](#h.z43g9coj7jns)

[modello delle specifiche](#h.5otgwwo95h6f)

[Design del simulatore next-event](#h.fegg3gd1we2j)

[Variabili di stato](#h.ly0whmg3mgni)

[Eventi](#h.53iaeufxjxf)

[Clock di simulazione e schedulazione degli eventi](#h.lvm4ldo0auf3)

[Calendario degli eventi](#h.dd4jfgbg3x6a)

[Design del modello di simulazione](#h.4zv5ixs7djvf)

[Variabili di inizio e termine simulazione](#h.65wnic76soq6)

[Calcolo indici di prestazione](#h.t0849hgrz2rv)

[Capitolo 4](#h.s044nlwjfatm)

[modello computazionale](#h.opxnmwjmtw0j)

[Calendario degli eventi](#h.ireo5k3vzkjt)

[Capitolo 5](#h.gs5bsejig7p2)

[test e verifiche](#h.z3ys8j32fqf9)

[Problematiche affrontate: monitoraggio dell’utilizzazione nel front-end Server](#h.2jphtzw0oa7f)

[Capitolo 6](#h.en9hyyakndi6)

[validazione del modello](#h.6kux64kohn54)

[Capitolo 7](#h.qseinxl07vv1)

[Analisi dei risultati](#h.p5r2bww1wdmk)

[sistema senza OVERLOAD MANAGEMENT](#h.80uvjk2hs64y)

[sistema con OVERLOAD MANAGEMENT](#h.wpmp6q6v2pn5)

[Intervalli di confidenza](#h.18kmnvchvdn6)

[Appendice B](#h.lfgppmrnjqng)

# PARTE 1

# Introduzione

Il progetto presentato si propone di testare il noto generatore pseudo-casuale di numeri random di Lehmer utilizzando, a tale scopo, uno dei sei test di casualità illustrati nella sezione 10.1[[1]](#footnote-0).  
Per una migliore comprensione sull’importanza dell’’uso di tale test, si è preferito implementare una propria versione del generatore lehmer; si è, quindi,scelto un moltiplicatore differente da quello proposta dal libro di testo che soddisfacesse dei requisiti particolari (spiegati nelle sezioni successive); una volta testato il generatore con tale moltiplicatore scelto effettuando i test preposti si sono confrontati i risultati riscontrati con quelli ottenuti dall’uso dei moltiplicatori e di altri parametri utilizzati dal libro di testo. Infatti da tale confronto si è compreso se le scelte effettuate a priori riguardo a una implementazione “nativa” possono considerarsi valide, e questo è possibile analizzando i grafici ottenuti in seguito alle simulazioni che forniscono indicazioni sul grado di casualità del generatore testato con i parametri di input scelti.

Per tali test sono stati utilizzati le librerie offerte dal libro di testo, i quali forniscono una lista di funzioni API (scritti in linguaggio C) pronte per essere utilizzati in qualsiasi programma simulativo.

Per sfruttare al maglio tali API, si è preferito implementare il progetto con lo stesso linguaggio di programmazione, mentre per la costruzione e visualizzazione dei grafici in seguito ai test simulativi è stato scelto di usare GNUPLOT per il linguaggio C.

# Capitolo 1 Implementazione del generatore Lehmer

## Scelta del moltiplicatore

Il primo passo necessario per una implementazione nativa del generatore Lehmer è stata la scelta di un moltiplicatore che godesse le proprietà di full-period e module-compatible rispetto al modulo m, dove m è il più grande numero intero primo rappresentabile in un calcolatore elettronico a 32 bit (m= 231 - 1). Si è dovuto, quindi, creare la lista di tutti i moltiplicatori che avessero simili proprietà e sceglierne uno tra questi per i nostri scopi. Il numero totale di moltiplicatori full-period e module-compatible calcolati in una architettura a 32 bit sono non più di 23093. Tra questi ci sono anche i moltiplicatori utilizzati dal libro di testo, ossia il moltiplicatore **16807**, usato largamente in passato[[2]](#footnote-1) fino alla scoperta del moltiplicatore ad oggi più utilizzato e considerato migliore in termini di generazione di sequenze di numeri random, il cui valore intero è **48271**. Di seguito verrà illustrato l’algoritmo utlizzato per il calcolo dei moltiplicatori full-period e module compatible rispetto ad m:

*while (x != 1){*

*if ((MODULUS % x < MODULUS / x) && gcd(i, MODULUS - 1) == 1){*

*/\* x is a full period multiplier equal to a^i mod m \*/*

*if (index < DIM){*

*arrayMultipliers[index] = x;*

*index++;*

*}*

*else break;*

*}*

*i++;*

*t = a \* (x % ((long)MODULUS / a)) - (MODULUS % a)\*(x / (long)(MODULUS / a));*

*if (t > 0)*

*x = t;*

*else*

*x = t + MODULUS;*

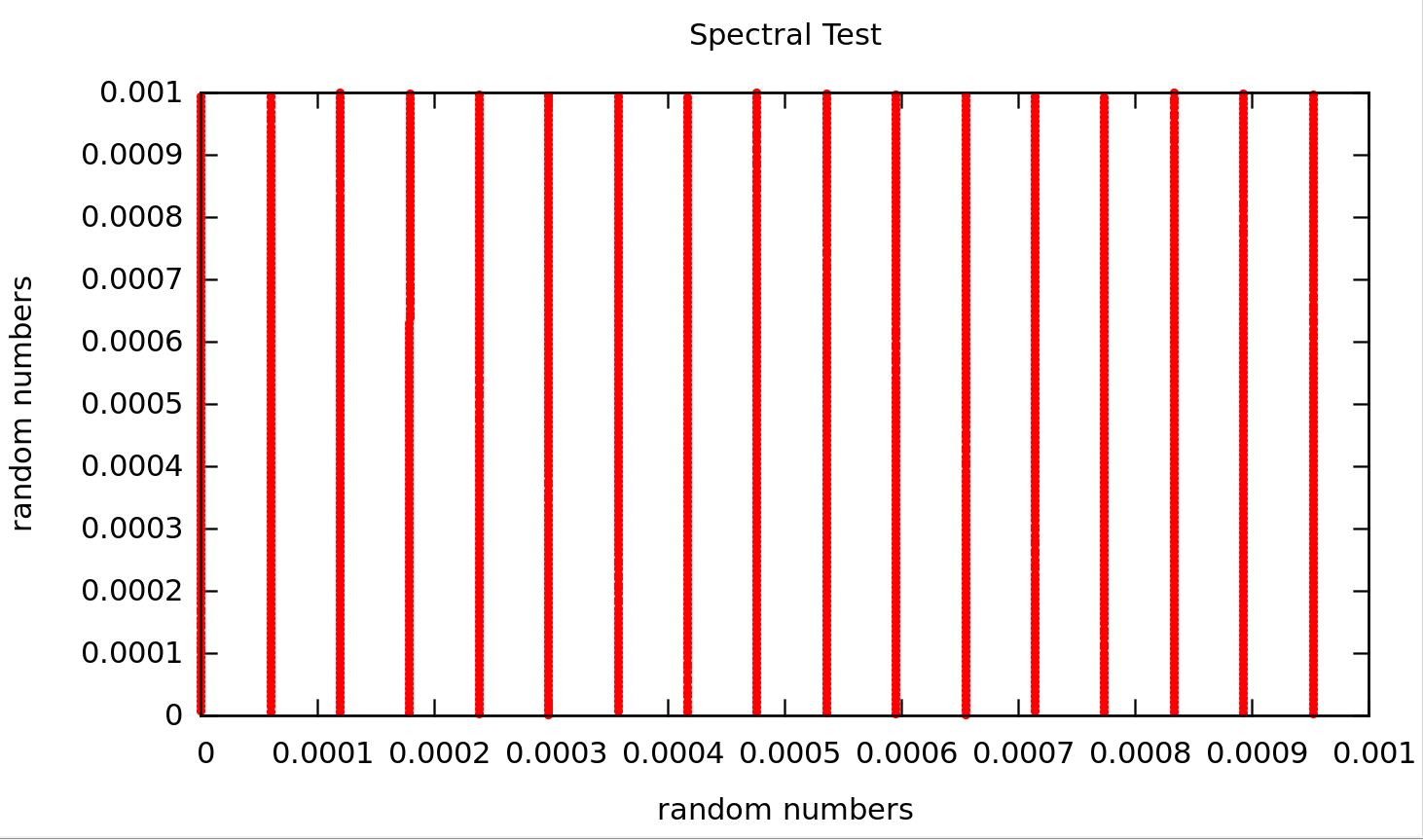
Dato il modulo primo m ed un qualsiasi moltiplicatore full-period e module -compatible, l’algoritmo riportato sopra genera tutti i moltiplicatori full period e module compatible a m. Per ogni iterazione dentro il while, viene controllato se il moltiplicatore *x* calcolato nella iterazione precedente soddisfa tali proprietà, in tal caso viene inserito in un’array, altrimenti si procede nel calcolo successivo di un nuovo intero. Tale iterazione è esprimibile come: *ai mod m*.

Per evitare condizioni di overflow dovuti al prodotto a\*Xi vengono calcolate resto e il quoziente della funzione, e in base al segno dell’operazione di sottrazione tra questi è possibile determinare il valore intero del moltiplicatore senza produrre alcun valore intermedio o finale più grande di m-1.

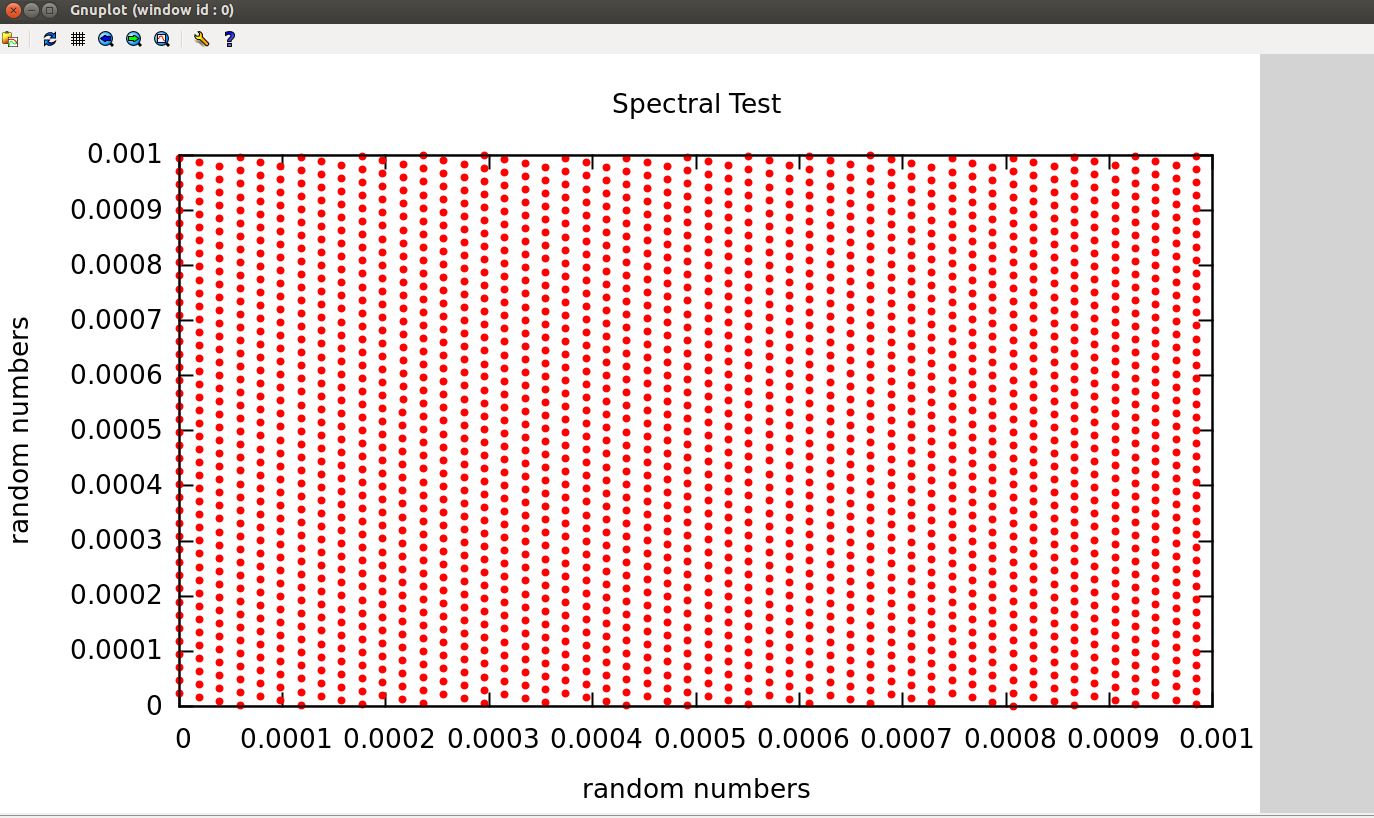
Dopo aver determinati tutti e 23093 moltiplicatori, questi vengono salvati in un file di testo per un facile recupero in un secondo momento. Per i nostri test abbiamo scelto infine il moltiplicatore di valore **50812** che gode delle proprietà di *FULL-PERIOD e MODULE-COMPATIBLE rispetto al modulo m*.

## Test spettrale

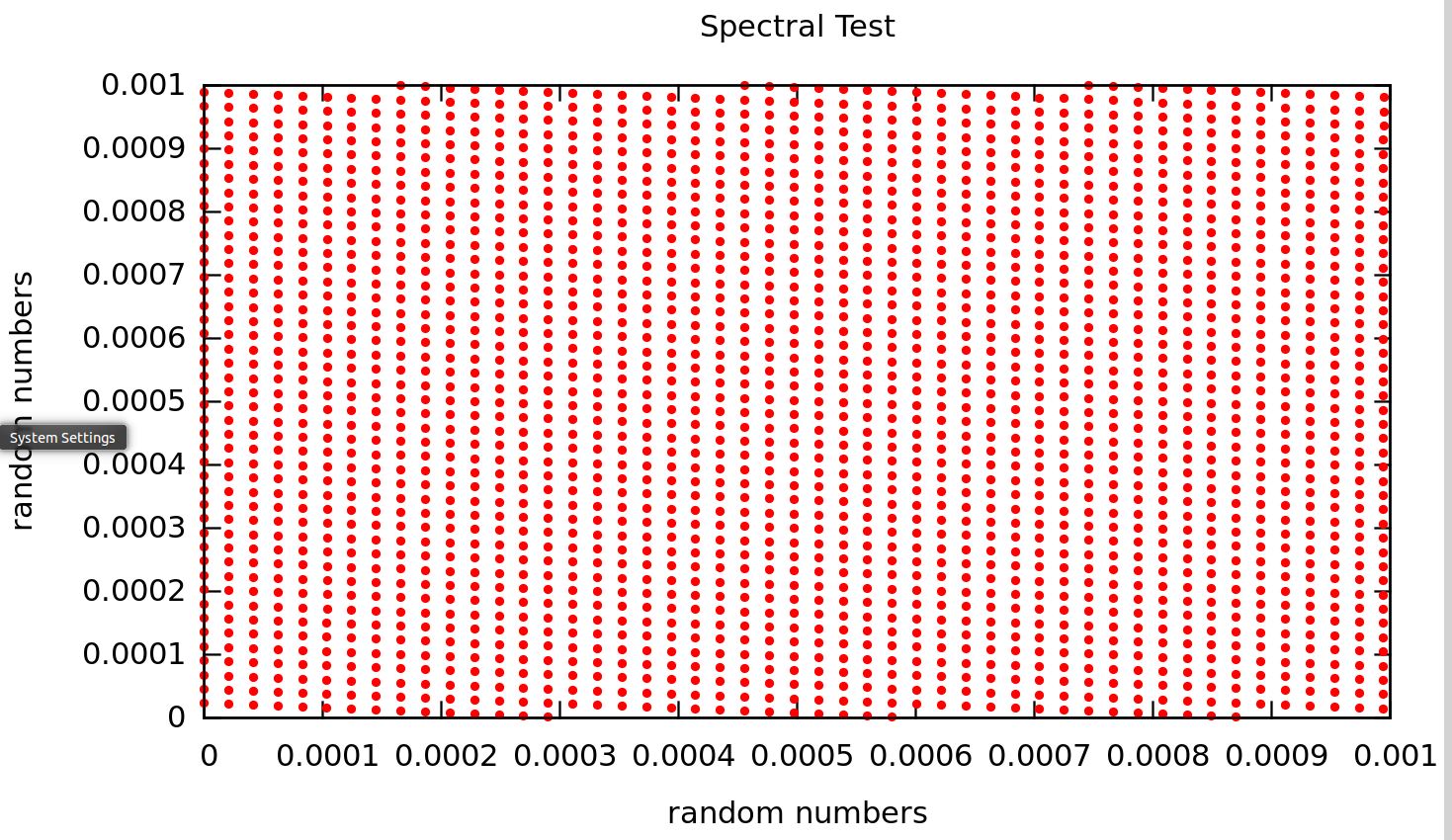
Il primo obiettivo che ci si è posti dalla scelta del moltiplicatore full-period e module-compatible rispetto al modulo m[[3]](#footnote-2), è di scegliere quel moltiplicatore in modo che il generatore di lehmer potesse generare una sequenza di numeri “il più casuale possibile”. Comunque non esiste un metodo generale per definire il grado di casualità di una sequenza di numeri generati, tuttavia è stato scelta una tecnica piuttosto popolare in cui si stabilisce una metrica del grado di casualità. Tale test è conosciuto con il nome di *Test Spettrale[[4]](#footnote-3).*  
Esso è basata sulle proprietà geometriche dei generatori Lehmer, in cui i valori generati vengono prese a coppie e rappresentati in punti in uno spazio bidimensionale (è possibile estendere anche in k-dimensioni). Allora per ogni ki punti formati dai k valori(coordinate) formano una struttura reticolata. La metrica di casualità sui generatori Lehmer si tratta quindi di analizzare l’uniformità di tali strutture reticolate in uno spazio a k-dimensioni per valori di k relativamente piccoli.  
Si è proseguito tale studio analizzando dapprima il generatore di lehmer (a,m)=(16807, 231 - 1) con il generatore scelto (a,m)=(50812, 231 - 1) e successivamente, con il generatore usato dal libro di testo (a,m)=(48271, 231 - 1). Il test è stato effettuato utlizzando lo stesso seme iniziale per tutti e tre i generatori, e sono stati considerati soltanto i punti compresi tra (0.0) e (0.001, 0.001) per una migliore analisi. Di seguito sono riportate i grafici relativi ai risultati delle simulazioni:



*test effettuato con multiplier a = 16807*



*test effettuato con multiplier a = 50812*



*test effettuato con multiplier a = 48271*

Dai grafici riportati sopra, risulta evidente come dallo zoom effettuato sulla struttura a reticolo delle coppie (Xi, Xi+1), per i numeri prodotti dal moltiplicatore a=16807 esiste una struttura non uniforme dei punti, ma a linee verticali (probabilmente dovuta a una correlazione seriale tra le coppie di valori), rispetto a quella relativa al moltiplicatore a=48271. Un generatore ideale non prevede una struttura ordinata a qualsiasi livello di zoom si volesse effettuare (reti scale-free o frattali), ma in questi casi abbiamo generatori random reali che si discostano da quello ideale a causa della presenza di tale ordine. L’uniformità dei punti da una indicazione sulla bontà di un generatore e il moltiplicatore a=48271 è considerato tra i migliori scoperti.

Il moltiplicatore a=50812 scelto per questo progetto presenta anch’esso una struttura abbastanza uniforme, senz’altro migliore della struttura relativa al moltiplicatore a = 16807. Pertanto si è scelto di utilizzare tale moltiplicatore per il proseguimento del progetto.

## Calcolo del Jump Multiplier

Finora i test spiegati in precedenza sia per determinare i moltiplicatori full-period e module-compatible con m che per i test spettrali si è generato un unico stream di valori pseudo-casuali. Tuttavia nei programmi di simulazione discrete-event in cui si vuole simulare un sistema composto da varie componenti stocastiche, è conveniente utilizzare un generatore multi-stream e, a tale scopo, viene fornita la libreria ***rngs.c*.**Tale libreria implementa la funzione di lehmer estesa al multi-stream, introducendo il moltiplicatore di salto: *aj mod m* che viene computata solo una volta; grazie a questo, infatti la funzione di salto gj (x) = (*aj mod m*)\*x mod m fornisce un modo per separare diverse sequenze di numeri random, e se il parametro di salto j viene scelto in modo appropriato, viene garantita anche la disgiunzione degli stream creati, ognuno con un proprio seme iniziale determinato dal primo seme iniziale impostato.

in questo progetto si è scelto di mantenere 256 streams di numeri random previsti dalla libreria ***rngs.c*.** Tuttavia, si deve considerare il nuovo moltiplicatore scelto, **a = 50812,** per cui è necessario determinare nuovamente la variabile di salto j tale che il moltiplicatore di salto   
*aj mod m sia modulo compatibile con m[[5]](#footnote-4).*

La variabile di salto j, che determina la partizione della sequenza in 256 streams disgiunti di uguale lunghezza, dovrà quindi essere il più grande intero minore di 231 / 28 =8388608 per garantire tale proprietà. L’implementazione per determinare il nuovo moltiplicatore di salto è riportato di seguito:

while(j < J)

{

t = a \* (jump % ((long)MODULUS / a))-(MODULUS %

a)\*(jump/(long)(MODULUS / a));

if(t > 0)

jump = t;

else

jump = t + MODULUS;

if(check(jump)){

result = jump;

jmax = j+1;

}

j++;

}

return result;  
  
La funzione prende in input il moltiplicatore di base a e imposta le variabili locali a zero. Per ogni iterazione viene computato il valore del moltiplicatore di salto utilizzando il confronto con il resto e il quoziente rispetto ad m in modo simile alla computazione dei moltiplicatori full-period e module-compatible visto nella sezione precedente (per evitare l’overflow del prodotto). Viene quindi controllato se il moltiplicatore di salto è module compatible con m tramite la chiamata alla funzione *check()*:

int check(long a)

{

long r;

long q;

long m;

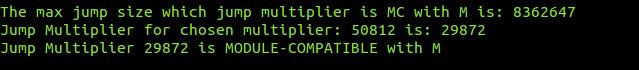
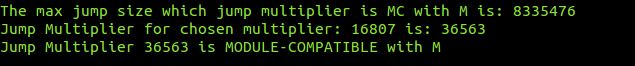
r = MODULUS % a;

q = (long)MODULUS / a;

m = a\*q + r;

if (r < q && m == MODULUS) return 1;

else return 0;

}  
  
in caso affermativo viene aggiornata la variabile *result* e *jmax* che rappresentano rispettivamente il moltiplicatore di salto finale restituita dalla funzione e la variabile di salto j. Scegliendo il moltiplicatore a = 50812 il risultato è stato riportato qui sotto:  
  
  
  
Inoltre è stato fatto un test simile anche per il moltiplicatore a = 16807 che non è riportato sul libro:  
  
  
mentre il test per a = 48271 è stato effettuato al solo scopo di verificare la correttezza dell’algoritmo, confrontando i risultati con quelli riportati nel libro di testo.  
  
una volta determinato il moltiplicatore di salto, è stato quindi reso necessario modificare la libreria ***rngs.c*** in modo da essere adatta ai scopi del progetto. Con i nuovi valori del moltiplicatore **a = 50812** e il moltiplicatore di salto **aj mod m = 29872** integrati nella libreria è possibile utilizzare il generatore multi-stream di Lehmer per i randomness test riportati nel capitolo 10 del libro di testo.

# Capitolo 2

## Extremes Test

L’ultima parte di questo progetto riguarda l’applicazione dei test di casualità per il generatore di numeri random. Il generatore Lehmer usato è preimpostato con *(a,m) = (50812, 231 -1)* nella libreria **rngs.c** spiegato nella sezione precedente, mentre il test utilizzato è stato il test degli estremi (extremes test)*.*  
Per la simulazione di tale test si può riassumere il processo in tre passi:

* Generazione di un campione di valori con chiamate ripetute al generatore.
* Computazione di un test statistico la cui distribuzione (pdf o funzione di densità di probabilità) è nota su variabili random uniformi in (0,1) indipendenti e identicamente distribuiti (*iid*).
* Valutare la verosimiglianza del valore computato del test statistico con la relativa distribuzione teorica da cui è stato assunto adottando una metrica basato sulla distanza lineare.

Il test degli estremi si basa sulla considerazione che se si ha una sequenza iid di variabili random uniformi in (0,1) e se una variabile R = max{U0, U1,..., Ud-1} allora la variabile random U = Rd è anch’essa uniforme in (0,1)[[6]](#footnote-5).  
  
l’algoritmo del test degli estremi effettua un raggruppamento (batching) dei valori estratti dal generatore in gruppi di uguale lunghezza (determinato dal parametro *d*), trovando il massimo di ogni batch, elevandolo tale massimo all d-esima potenza e conteggiando tutti i massimi generati in un array come illustrato di seguito:  
  
void Initialize(void)

{

int x;

int i;

int j;

int d;

double u;

d = 5;

double r;

for(x = 0; x < K; x++)

o[x] = 0;

for(i=0; i < N; i++){

r=Random();

for(j=1; j < d; j++){

u = Random();

if(u > r){

r = u;

}

}

u = exp(d \* log(r));

x = (int)(u \* K);

o[x]++;

}

}  
  
Tale funzione viene applicata per ogni stream del generatore Lehmer nel modo riportato di seguito:  
  
for(i=0; i < 256; i++){

SelectStream(i);

Initialize();

v = Chi\_Stat();

chi[i] = v;

Per ogni stream, si determina quindi la variabile chi-quadro *v*  tramite la chiamata della funzione *Chi\_Stat():*   
double Chi\_Stat(double v1, double v2)

{

int x;

double sum;

double num;

double den;

num=0.0;

den=0.0;

sum=0.0;

for(x=0; x < K; x++){

num = pow((o[x] - N/K), 2);

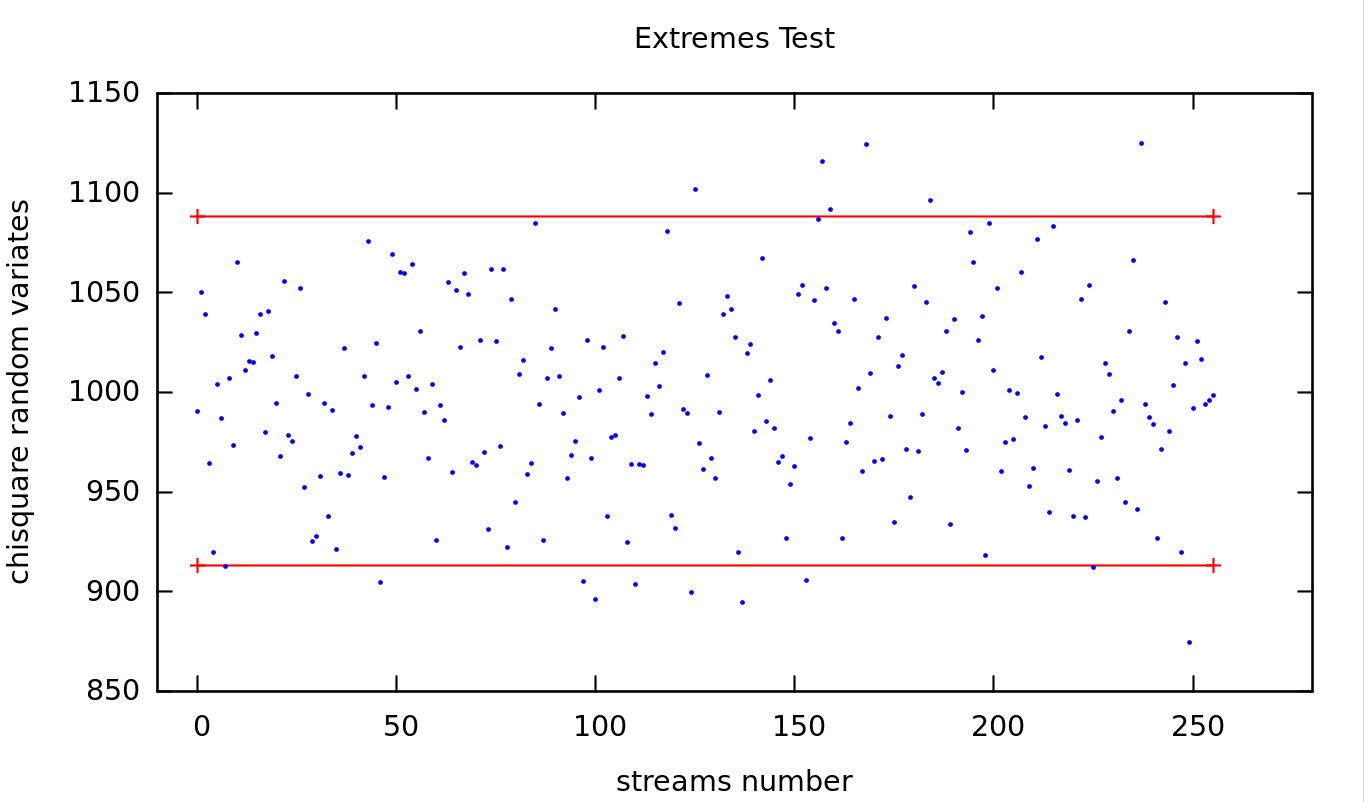
den = N/K;

sum += num/den;

}

return sum;

}

I valori critici v\*1 e v\*2 vengono calcolate utiizzando la funzione inversa *idfChisquare(long n, double u)*  fornita dalla libreria ***rvms.c*** del libro di testo.Bisogna precisare che per il calcolo di tali variabili statistiche è stato scelto un livello di confidenza con parametro alpha = 0.05, mentre i parametri N e K sono rispettivamente N = 10000 e K = N/10 = 1000.  
Successivamente si confronta la statistica chi quadro *v,* determinata al passo precedente, con i valori critici v\*1 e v\*2 , per ogni stream(in totale sono 256 variabili chi-quadro *v* ).  
Se v < v\*1 o v > v\*2 il test fallisce (per quello stream) con probabilità 1 - alpha.  
Il grafico risultante di questo test empirico è illustrato di seguito:  
  
  
 *test degli estremi con K=1000, N=10000 e d=5*

I valori critici sono visualizzati come linee rosse orizzontali: quella inferiore rappresenta  
v\*1 = 913.3 mentre quella superiore è v\*2 = 1088.5  
  
Dalla simulazione effettuata si è notato che il numero di test statistici v > v\*2 sono stati 6 mentre il numero di test v < v\*1 sono stati 10. Di seguito è riportato l’output del programma:  
  
  
  
Considerando il numero totale di test falliti (upper e lower bound)[[7]](#footnote-6) pari a 16, si nota che non ci si discosta molto rispetto al valore atteso approssimato; infatti in 256 test con un livello di confidenza del 95% il valore aspettato è circa 256 \* 0.05 = 13 fallimenti. Questo valore può essere una indicazione della bontà del generatore di Lehmer implementato.

## Confronto con moltiplicatore 48271 e 16807

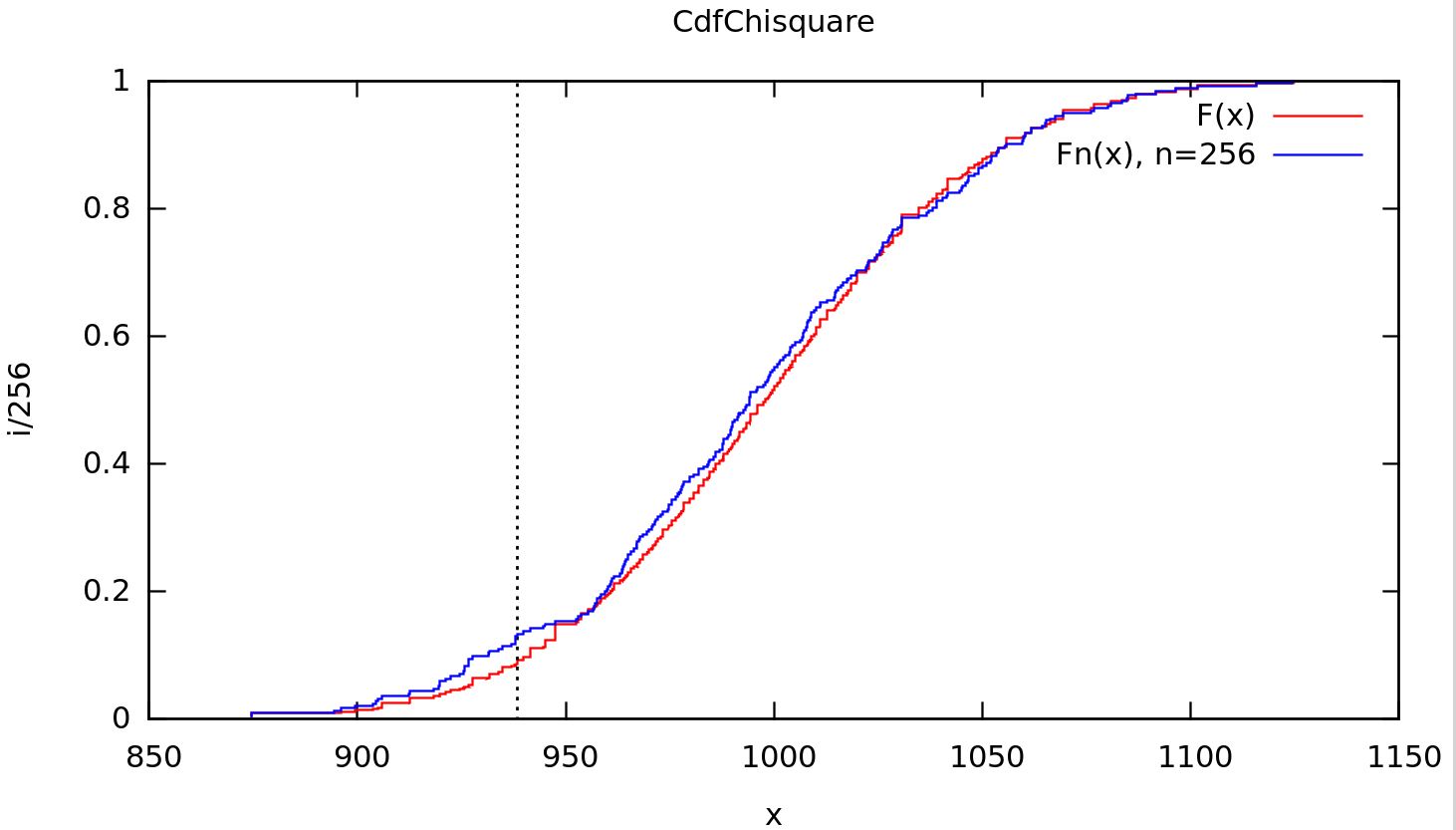
dai risultati visti in precedenza si è notato che il numero totale di test falliti è pari a 16 non lontano dal valore di riferimento pari a 13.

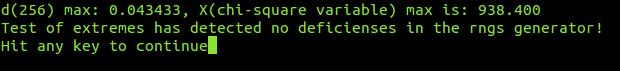
### Analisi di Kolmogorov-Smirnov

Il test statistico di Kolmogorov-Smirnov misura la più grande distanza verticale tra la funzione di distribuzione cumulativa calcolato da un dataset e una funzione di distribuzione cumulativa teorica[[8]](#footnote-7). tale statistica è indicata con *d.*

Il primo passo per effettuare questo tipo di test è di ordinare la sequenza dalle variabili del dataset in senso crescente applicando un algoritmo di ordinamento  
Se n=256 e alpha=0.05 si ha quindi d\*=0.084. Perciò la probabilità che una singola variabile KS d256 sia minore di 0.084 è di 0.95. Il test KS può considerarsi fallito se la statistica computata d256 supera il valore critico d\*=0.084.  
La simulazione di questo test è stato effettuato con il generatore Lehmer settato con i parametri *(a,m) = (50812, 231 -1)*  n=256 ed alpha=0.05*.*  
Dal risultato della simulazione il valore d256 computato ha il valore pari a d256 = 0.043433 minore del valore critico d\*=0.084. pertanto si può considerare il test KS superato con successo.

Il grafico ottenuto viene riportato di seguito:

  
 confronto tra la cdf teorica e la cdf computata

la linea verticale tratteggiata indica il valore della statistica chi-quadro a cui è stato determinato la distanza massima. L’output del programma è riportato qui sotto:  
  


# Conclusioni

La prima parte di questo progetto dimostra come la scelta di implementare un proprio generatore random comporta tutta una serie di considerazioni (e riflessioni) necessarie se si vuole avere un grado di bontà accettabile riguardo alla generazione di sequenze pseudocasuali. Infatti pur scegliendo un proprio moltiplicatore che soddisfa le proprietà discusse nei capitoli precedenti e il proprio generatore riesca a passare i test statistici introdotti, tale generatore random potrebbe risultare persino peggiore tra quelli conosciuti (basti vedere il confronto fatto dal libro con i moltiplicatori **a=16807** con **a=48271**)**.** Per questo motivo esiste la differenza tra generatori ideali e quelli reali

# *PARTE 2* Simulatore di un sistema web multi-tier

# 

# Introduzione

In questa sezione si descrive la seconda del progetto, che consiste nella modellazione e nella valutazione delle prestazioni di un sistema multi-tired che fornisce servizi web. A tale scopo è stato realizzato un simulatore ad eventi discreti di tipo next-event, seguendo un approccio iterattivo.

Il modello così ottenuto è stato successivamente adottato per la valutazioni delle prestazioni del sistema web considerato in base alle specifiche fornite. Tale valutazione è stata effettuata variando le distribuzioni dei tempi di servizio del front-end, e implementando successivamente un meccanismo di “gestione del carico” basato sul monitoraggio dell’utilizzazione della CPU del front-end.

Per l’implementazione di tale simulatore si è preferito utilizzare un linguaggio di programmazione general-purpose come Java.

# Capitolo 1

## Obiettivi

Il primo passo eseguito è stato quello di individuare gli obiettivi dello studio da condurre sulle prestazioni del sistema web considerato. Dalle specifiche fornite, infatti, è emersa la necessità di analizzare le prestazioni del sistema variando le distribuzioni del tempo di servizio del front-end server, in base ai seguenti indici di prestazione:

* Tempo di risposta del sistema
* Throughput del sistema
* drop ratio e abort ratio

I risultati ottenuti sono stati confrontati tra loro con l’obiettivo di individuare il sistema con le peggiori prestazioni.  
  
Successivamente si sono valutate le prestazioni del sistema ”peggiore” adottando un meccanismo di overload management. Questo meccanismo si basa sul monitoraggio della utilizzazione del front-end server, il quale si occupa di ricevere le richieste sia di nuove sessioni che di quelle online (richieste pendenti).

# Capitolo 2

## Modello concettuale

Il sistema considerato consiste di un front-end server (con relativa coda), il quale si occupa di ricevere sia le richieste di nuovi sessioni che di quelle già presenti nel sistema stesso (richieste online), e di un back-end server (con relativa coda), il quale riceve le singole richieste provenienti dal front-server. Le sessioni arrivano al front-end ad istanti di tempo random.  
Una sessione consiste di un insieme di singole richieste di un utente. Dopo che una sessione è stata accettata dal sistema, le singole richieste della stessa sessione circolano nel front-end e back-end server richiedendo servizio, prima di ritornare al client. Anche ill tempo di servizio è random. Una volta che la richiesta ritorna al client, esso spende un tempo di thinking random prima di inviare una nuova richiesta. La sessione viene completata dopo che tutte le richieste di quella sessione sono state elaborate. Di seguito viene illustrato il diagramma del sistema:  
  
   
 *Figura 1: modello del sistema*

Il sistema è stato modellato assumendo una capacità infinita per le code del front-end e back-end server, e con disciplina di coda *FIFO*. Inoltre il servizio è senza prelazione e conservativo. Per quanto riguarda iI client del centro delle sessioni attive si è adottato un infinite server..

Di seguito vengono elencate le variabili di stato rilevanti del modello in esame:

### *Variabili di stato*

* numero di richieste in servizio al tempo t nel front-end e nel back-end
* numero di richieste nel front-end server e nel back end server al tempo t
* il numero di sessioni attive nel centro terminali.

# Capitolo 3

## modello delle specifiche

Nello sviluppo del modello delle specifiche, innanzitutto l’attenzione è stata rivolta alla definizione dei modelli di input da utilizzare nel modello di simulazione. Tali modelli sono stati definiti in base alle specifiche fornite nel seguente modo:

* tempo di servizio del front-end, distribuito *esponenzialmente* con media 0,00456 s
* tempo di servizio del back-end, distribuito *esponenzialmente* con media 0,0017 s
* tempo di interarrivo delle nuove sessioni, distribuito *esponenzialmente* di parametro 35 sessioni/s
* lunghezza delle sessioni, *uniforme discreta* di parametri a=5, b=35.
* tempo di thinking, distribuito *esponenzialmente* di parametro 7 s.

Pertanto si è scelto di sviluppare un simulatore distribution-driven usando un approccio next-event che permette di gestire facilmente l’avanzamento del tempo simulato, e che si adatta bene al crescere della complessità del sistema.

## Design del simulatore next-event

Nella costruzione del modello di simulazione next-event sono stati eseguiti principalmente 3 step:

* costruzione delle variabili di stato, che forniscono una descrizione completa del sistema
* identificazione degli eventi
* costruzione di un insieme di algoritmi che definiscono i cambiamenti di stato che devono essere eseguiti all’occorrenza di un dato evento

Di seguito illustriamo gli elementi sopra essenziali del modello di simulazione next-event..

### Variabili di stato

* nfront-end(t): numero dei job nel front-end
* nback-end(t): numero dei job nel back-end server
* xfront-end(t): stato di occupazione del front end server
* xback-end(t): stato di occupazione del back-end server
* nsessions(t): numero delle sessioni attive

Altre variabili sono state utilizzate al fine di calcolare le statistiche (medie, varianze, ecc.), che verranno illustrate nel modello computazionale

### Eventi

Gli eventi identificati che possono cambiare lo stato del sistema sono:

* ***Completation front end****:* si verifica al completamento di una richiesta nel front-end server;
* ***Completation back end****:* si verifica al completamento di una richiesta nel back-end server;
* ***New Session****:* si verifica alla creazione di una nuova sessione;
* ***New request****:* si verifica alla generazione di una nuova richiesta.

### Clock di simulazione e schedulazione degli eventi

Per mantenere traccia dell’avanzamento del tempo simulato, è stata introdotta una variabile *clock*. Il meccanismo di avanzamento del tempo scelto è il *next-event time advance*, in cui gli eventi vengono processati in ordine crescente rispetto al tempo di schedulazione.

### Calendario degli eventi

Gli eventi vengono salvati in una struttura dati, salvando per ogni evento il tempo di occorrenza dell’evento stesso.

  
*Figura 2: struttura dati del calendario degli eventi (coppie <key,value>)*

## 

## Design del modello di simulazione

In questa sezione si descrive la struttura del modello di simulazione adottato, specificando sia le componenti più importanti che lo caratterizzano che le interazioni di tali componenti riportando i flussi di esecuzione.  
  
*Arrivo di una nuova sessione al tempo* t

* Schedula prossimo evento di tipo new session e aggiorna il calendario
* Estrai il numero delle richieste della sessione
* invia la richiesta al front-end server
  + se front-end è libero esegue la richiesta e aggiorna il calendario degli eventi relativo all’evento *completation front-end*.
  + altrimenti inserisci la richiesta in coda

*Completamento di una richiesta nel front-end server*

* Controlla se la coda del front-end server è vuota
  + se è vuota, aggiorna il calendario settando il tempo di occorrenza dell’evento completation front end a INFINITY (per denotare l’occorrenza dell’evento impossibile).
  + altrimenti estrai la richiesta dalla coda e schedula l’occorrenza dell’evento completation front server aggiornando il calendario
* Invia la richiesta completata dal front-end server al back-end server
  + se back-end è libero esegue la richiesta e aggiorna il calendario degli eventi relativo all’evento *completation back-end*.
  + altrimenti inserisci la richiesta nella coda del back-end server

*Completamento di una richiesta nel back-end server*

* Controlla se la coda del back-end server è vuota
  + se è vuota, aggiorna il calendario settando il tempo di occorrenza dell’evento completation back-end a INFINITY (per denotare l’occorrenza dell’evento impossibile).
  + altrimenti estrai la richiesta dalla coda e schedula l’occorrenza dell’evento completation back-server aggiornando il calendario
* Controlla se la sessione relativa alla richiesta completata è completa
  + se la sessione relativa è completa aggiorna il numero delle sessioni attive
  + altrimenti invia la richiesta ai terminali e aggiorna il calendario degli eventi relativo all’evento *new\_request*.

*Nuova richiesta al tempo t*

* Schedula prossima occorrenza dell’evento *new request*
* invia la richiesta al front-end server
  + se front-end è libero esegue la richiesta e aggiorna il calendario degli eventi relativo all’evento *completation front-end*.
  + altrimenti inserisci la richiesta in coda del front-end server

*Schedulazione degli eventi*

1. **Inizializzazione**
   1. Inizializza il clock di simulazione
   2. Imposta l’occorrenza dell’evento *completation front-end* a INFINITY
   3. imposta l’occorrenza dell’evento *completation back-end* a INFINITY
   4. imposta l’occorrenza dell’evento *new request* a INFINITY
   5. schedula l’occorrenza dell’evento *new session*
2. ***Processa l’evento corrente***

*Finchè il tempo di simulazione non è inferiore a STOP*

* 1. *scansiona la lista per determinare il prossimo evento che deve occorrere*
  2. *aggiorna il clock di simulazione al tempo di occorrenza di tale evento*
  3. *esegui l’algoritmo relativo a tale evento*

1. ***Terminazione***

### Variabili di inizio e termine simulazione

Per stabilire l’inizio e il termine della simulazione sono stati utilizzati variabili denominati: START, STOP e INFINITY.

la variabile START indica l’indica l’inizio della simulazione. Pertanto nella fase di inizializzazione descritto sopra il clock di simulazione viene settato al valore di questa variabile.

La variabile STOP viene utilizzata come condizione terminale di simulazione.

La variabile INFINITY viene invece adottata per indicare l’occorrenza di un evento come impossibile. Il valore viene impostato secondo la formula:

### Calcolo indici di prestazione

Il simulatore qui utilizzato genera un insieme di statistiche che permettono di ricavare informazioni utili per comprendere il comportamento del sistema.

Gli indici di prestazione calcolati sono:

* **Useful Throughput**  
  Lo useful throughput indica il numero di sessioni completate dal sistema in una unità di tempo:
* **Tempo di risposta del sistema**Il tempo di risposta del sistema indica il tempo che intercorre tra l’istante in cui una richiesta entra nel front-server e l’istante in cui la stessa esce dal back-end server
* **Aborted ratio**L’aborted ratio indica la percentuale di richieste rifiutate dal sistema rispetto al totale delle richieste.
* **Drop ratio**Il drop ratio indica la percentuale delle sessioni rifiutate dal sistema rispetto al totale delle sessioni

# Capitolo 4

## modello computazionale

### Calendario degli eventi

Per il calendario degli eventi è stato adottato una struttura dati complessa tipo  *HashMap<key,value>,* in cui come chiavi vengono considerati gli eventi mentre i valori sono i tempi di occorrenza di tali eventi:

public Calendar() {

calendar = new HashMap<Events, Double>();

calendar.put(Events.NEW\_SESSION, Main.INFINITY);

calendar.put(Events.COMPLETION\_FRONT\_SERVER, Main.INFINITY);

calendar.put(Events.COMPLETION\_BACKEND\_SERVER, Main.INFINITY);

calendar.put(Events.NEW\_REQUEST, Main.INFINITY);

calendar.put(Events.MONITORING, Main.INFINITY);

}

L’estrazione dell evento prossimo ad occorrere si basa sulla ricerca del minimo tempo effettuato sulla mappa, confrontando i tempi con il valore minimo:  
  
public Events getNextEvent(){

Iterator e = calendar.entrySet().iterator();

double min = Main.INFINITY;

Events nextEvent = null;

while(e.hasNext()){

Map.Entry pair = (Map.Entry)e.next();

if ((double)pair.getValue() < min){

min = (double) pair.getValue();

nextEvent = (Events) pair.getKey();

}

}

return nextEvent;

}

private void monitoringUtilization(Server frontEnd, Calendar calendar) {

calendar.setTimeEvent(Events.MONITORING, t.getCurrent() + sizeMonotoringWindow);

double utilization= frontEnd.getArea().getIstantAreaService() / (t.getCurrent() - resetTime);

resetTime=t.getCurrent();

frontEnd.getArea().resetIstantService(t.getCurrent());

if(reject && utilization < 0.75){

reject = false;

}

if(!reject && utilization > 0.85){

reject = true;

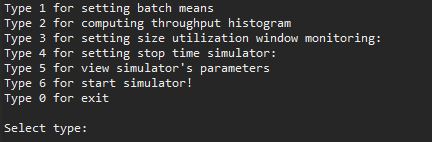
}

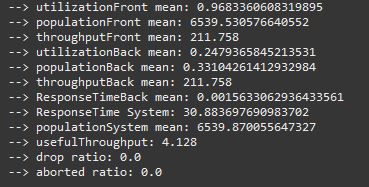
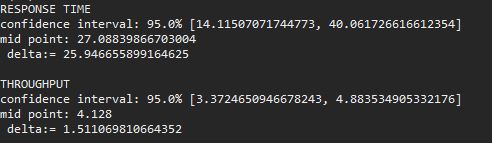
}

I dettagli del modello computazionale sono riportati nel appendice B,

# Capitolo 5

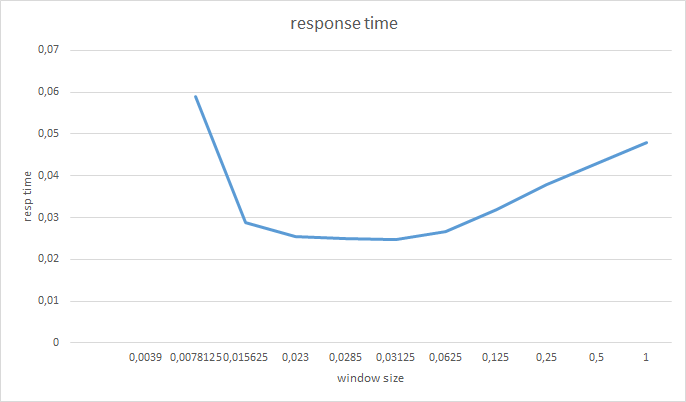
## test e verifiche

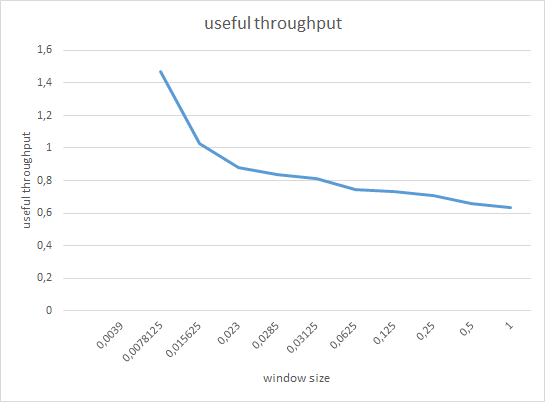
Le simulazioni effettuate per ottenere le statistiche sono state effettutate su macchine con sistema operativo Windows 7, mentre per il tracciamento di grafici è stato usato un tool esterno di microsoft, excel 2013. E’stata condotta un ampia attività di verifica al fine per controllare la corretta implementazione del modello computazionale, basata sia su controlli di consistenza, che confrontando l’output del simulatore con i risultati teorici ottenuti simulando il sistema in condizione stazionarie (4 sessioni al secondo, anzichè 35).   
Per lanciare il simulatore, basta avviare il Main del progetto utilizzando la console da eclipse e selezionare un elemento dal menu iniziale del simulatore:  
  


Quando si vuole effettuare un run di simulazione al termine di un potrà vedere una schermata contenente i valori relativi ai parametri del simulatore:. Un esempio viene illustrato di seguito (caso sistema senza overload management):  
  
  
  


### Monitoraggio dell’utilizzazione nel front-end Server

Uno dei problemi affrontati nella fase di test è stato quello relativo al monitoraggio dell’ utilizzazione del front end server.  
Inizialmente, mentre si eseguivano test sul sistema con il *meccanismo di overload management* per verificare l’esistenza della stazionarietà delle statistiche di interesse quali throughput e tempi di risposta, aumentando la durata dei run simulativi si notava che tali statistiche divergevano anche quando l’utilizzazione del front-end oscillava tra l’85% e il 75%. Secondo il meccanismo dell’overload management, quando l’utilizzazione nel front end arriva a 85% il sistema “chiude le porte ” sia alle nuove sessioni che alle richieste pendenti, per poi riaprirle quando l’utilizzazione scende sotto il 75%. Durante la fase di chiusura porte, la dimensione della coda del front server viene svuotata rapidamente per poi tornare a crescere quando il sistema è in fase di riapertura porte (ossia quando l’utilizzazione del front-end risale da 75% a 85%). Quando l’utilizzazione arriva a 85%, al momento della chiusura delle porte la dimensione della coda risulta essere molto più grande rispetto alla fase precedente di chiusura porte (quando l’utilizzazione era a 85%). A causa di questo, il tempo di risposta del front-end server (e quindi del sistema) tende ad assumere valori sempre più grandi aumentando la durata della simulazione, impedendo in questo modo un comportamento convergente a un valore stazionario della statistica stessa.  
Successivamente si è scoperto che la causa di tale comportamento era dovuto a una diversa temporizzazione delle misurazioni, dato che il monitoraggio dell’utilizzazione veniva effettuato dall’istante zero all’istante di monitoraggio,e questo determinava una minore reattività del sistema alle variazioni dell’utilizzazione: si è notato infatti che l’utilizzazione del front-end server aumentava sempre più lentamente ad ogni fase di chiusura e riapertura porte, mentre la coda diveniva sempre più grande.  
La soluzione a questa problema è stata quindi di introdurre un controllo sull utilizzazione ad intervalli di tempo prefissati. Resta da risolvere la questione sulla scelta ottimale della ampiezza della finestra di monitoraggio. Sono quindi state fatte dei test cambiando di volta in volta l’ampiezza delle finestre temporali.   
Il grafici seguenti mostrano l’andamento dei tempi di risposta del sistema, dello useful throughput e drop e abort ratio al variare della dimensione della finestra di osservazione:

  
*Figura 3: response time al variare della size di osservazione dell utilizzazione*

  
 *Figura 4: useful throughput al variare della size di osservazione dell utilizzazione*

# 

*Figura 5: response time al variare della size di osservazione dell utilizzazione*

L’unica questione da risolvere è determinare il valore ottimale della dimensione della finestra di osservazione che faccia da trade-off ai valori del tempo di risposta, del throughput e del drop/abort ratio. Infatti se si preferisce scegliere una dimensione della finestra di osservazione che favoriva la diminuzione dei tempi di risposta del sistema, si andava però a peggiorare la misurazione del throughput e/o del drop-abort ratio e viceversa.

Dopo varie discussioni la scelta è stata infine quella di scegliere quella dimensione ottimale della finestra temporale che minimizza i tempi di risposta e il valore della dimensione della finestra è pari a **0,0285 secondi.**

# Capitolo 6

## Validazione del modello

Per esaminare la consistenza del simulatore con il sistema analizzato, sono stati effettuate una un insieme di considerazioni, basate su un modello analitico semplificato a rete aperta (rete aperta di Jackson).

### Modello semplificato senza overload management



Si riportano di seguito le equazioni per la rete di jackson:

1=y+2(1-p)

1=2

con soluzioni

1= 2= y / p = 700 richieste/s.

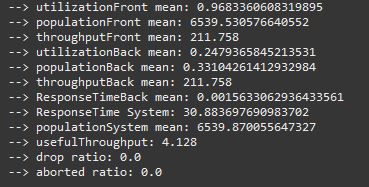
Dato che 1>>u1 e u2>>1 allora 2=u1

In seguito a questi risultati ottenuti, l’output steady state del simulatore senza meccanismo di overload dovrebbe essere consistente con i seguenti valori:

Ub = u1 / u2 =0.257

Xb = u1=219,29

E(Ts)b = E(S)b / (1 - u2)=0,001573



successivamente è stato considerato anche un modello semplificato con overload management (chiuso) con centro terminali:

### modello semplificato con overload management

Per convalidare il sistema con overload management si è preferito un approccio operazionale considerando un valore di N (numero medio di sessioni attive nel sistema) misurato direttamente dal simulatore e confrontando il throughput ottenuto dal simulatore con l’analisi asintotica ricavato dalle formule analitiche :

X = min {1 / Dmax  , N / (Dfront-end+Dback-end+E(Z))}

N = 639 sessioni attive

X = min {219,29; 91.2} = 91,2 mentre il valore Xsimulato = 90,63 rientrando nei limiti asintotici.

Tale confronto è stato effettuato anche per i valori dei tempi di risposta ed ottenendo anche in questo caso un tempo di risposta che rientrava nei limiti asintotici.

Ulteriori controlli di consistenza sono stati effettuati utilizzando la legge di Little.

# 

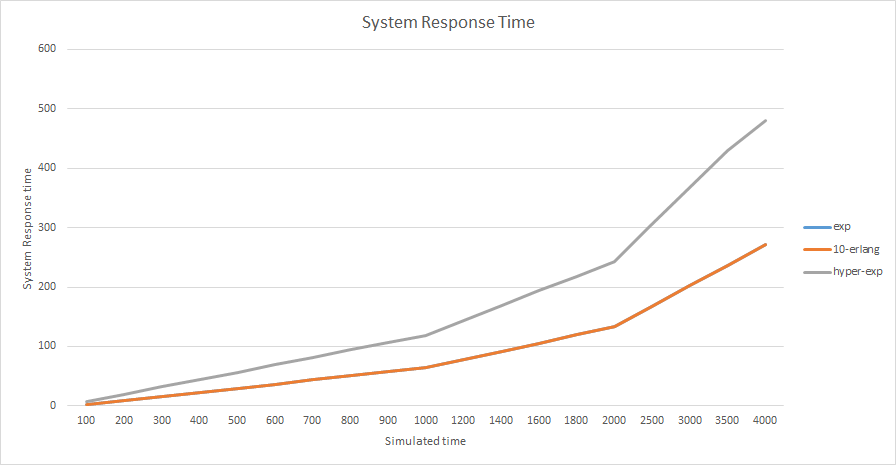
# Capitolo 7

## Analisi dei risultati

I dati relativi agli indici di prestazione di interesse(tempi di risposta e throughput) misurati nelle precedenti fasi di test sono stati in seguito aggregati e visualizzati in forma di grafici. Infatti, in base al tipo di test effettuato, tali grafici si suddividono in due categorie: i grafici di throughput e tempi di risposta del sistema in stato di instabilità (front server sovraccarico) e quelli relativi al sistema in condizione di stazionarietà (front server non sovraccarico).

### sistema senza OVERLOAD MANAGEMENT

In questo scenario l’utilizzazione del front-end server è praticamente uguale a 1, perciò il front-end server non riesce a completare tutte le richieste entranti, con la inevitabile situazione di vedersi aumentare indefinitamente la lunghezza della sua coda. Di conseguenza il *tempo di risposta del sistema* tende a divergere all’aumentare del tempo di simulazione. Il grafico illustrato di seguito mostra tale scenario:

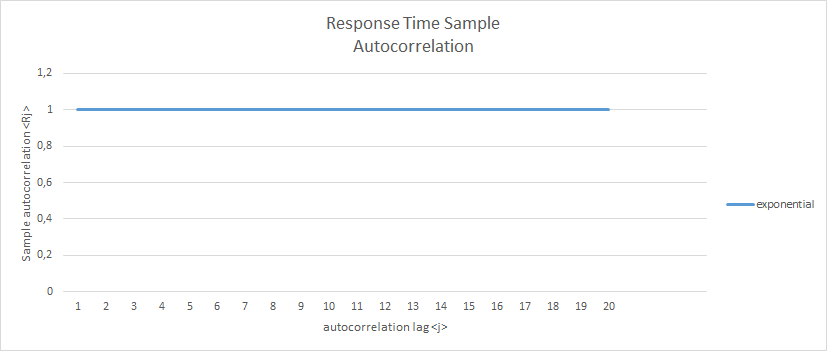
  
*Figura 6: tempi di risposta del sistema instabile*

In questi test sono stati effettuati diversi run aumentando per ogni run l’istante di STOP di simulazione e, alla fine di ogni run si salvavano le statistiche di interesse.  
Come si intuisce dal grafico sopra riportato, i tempi di risposta si contraddistinguono in base al tipo di distribuzione dei tempi di servizio del front-end server (iperesponenziale, 10-erlang ed esponenziale), e in base all’andamento delle curve dei tempi di risposta, risulta evidente come la distribuzione iperesponenziale dei tempi di servizio del front-end risulta peggiore che nel caso esponenziale e 10-erlang.

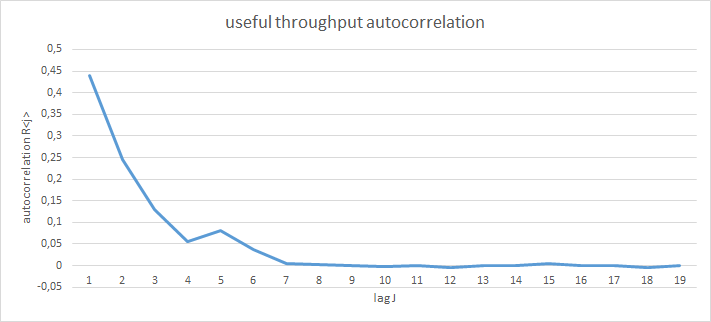
Questa differenza notevole è dovuta al fatto che, nel caso iperesponenziale, avendo preimpostato la probabilità **p=0,1** la varianza dei tempi di servizio delle richieste risulta essere molto elevato con la conseguenza di rallentare notevolmente il front-end server.

Di conseguenza la curva dei tempi di risposta della iper-esponenziale diverge più rapidamente rispetto alle controparti 10-erlang ed esponenziale. Tuttavia il grafico mostra anche un aspetto insolito: la curva del tempo di risposta della 10-erlang coincide praticamente con la curva dell’esponenziale anche avendo impostato un parametro **K=10,** mentre ci si aspettava al contrario un miglioramento dei tempi di risposta rispetto alla curva della esponenziale**.** Probabilmente la scelta del parametro **K** pari a 10 è insufficiente a garantire un miglioramento significativo, si propone perciò di verificare tale miglioramento aumentando di poco il valore di **K,**  senza aumentarlo considerevolmente o si comporterebbe come una distribuzione deterministica.

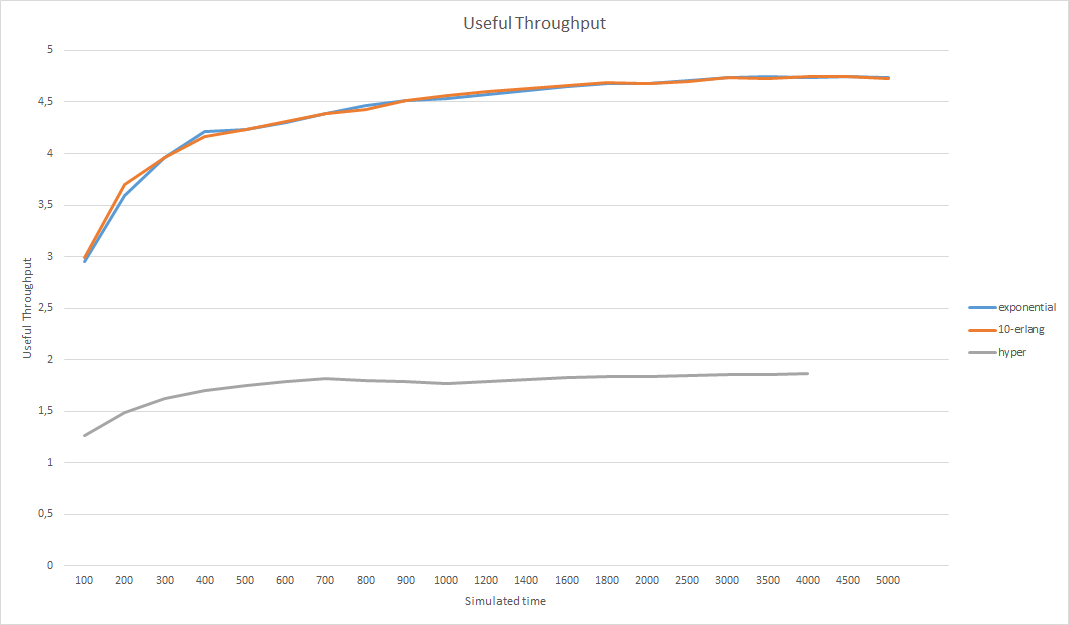
L’evidente divergenza dei tempi di risposta del sistema è evidenziato anche dalla forte correlazione presente dai tempi di risposta delle singole richieste presenti nel sistema.Il grafico seguente illustra tale correlazione sulla distribuzione esponenziale pari a **1** (identico anche nelle altre disribuzioni):

  
*Figura 7: autocorrelazione dei tempi di risposta*

mentre il grafico relativo all autocorrelazione dello useful throughput viene riportato di seguito:

  
*Figura 8: autocorrelazione dello useful throughput*

Il grafico successivo mostra l’andamento dello *useful throughput* i cui valori sono stati misurati dagli stessi test usati per il tempo di risposta del sistema. Anche in questo caso si distinguono le diverse distribuzioni dei tempi di servizio del front-end server:

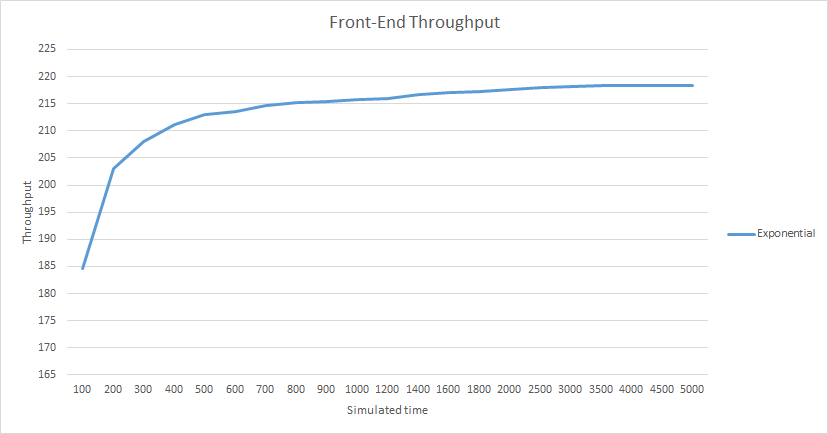


*Figura 9: useful throughput del sistema instabile*

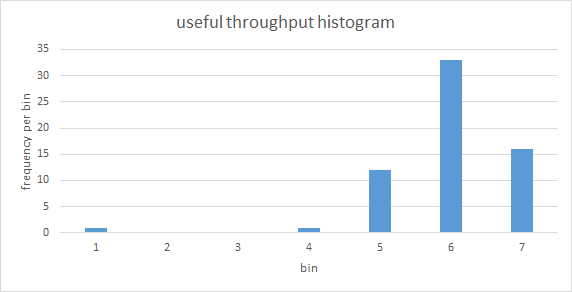
Anche in questo caso si nota la notevole differenza dello useful throughput del caso iperesponenziale da quelli esponenziali e 10-erlang, i quali coincidono per valori di run simulativi molto alti.

Lo useful throughput è l’unico indice di prestazione che presenta un andamento stazionario all’aumentare del tempo di simulazione. Si può notare dal grafico, infatti, che tale valore di stazionarietà è all’incirca pari a **4,73** sessioni completate per unità di tempo(run).

Il grafico illustrato di seguito mostra,, inoltre la convergenza del throughput del front-end server misurato per ogni richiesta completata dal server stesso:

  
*Figura 10: throughput del front-end per richieste*

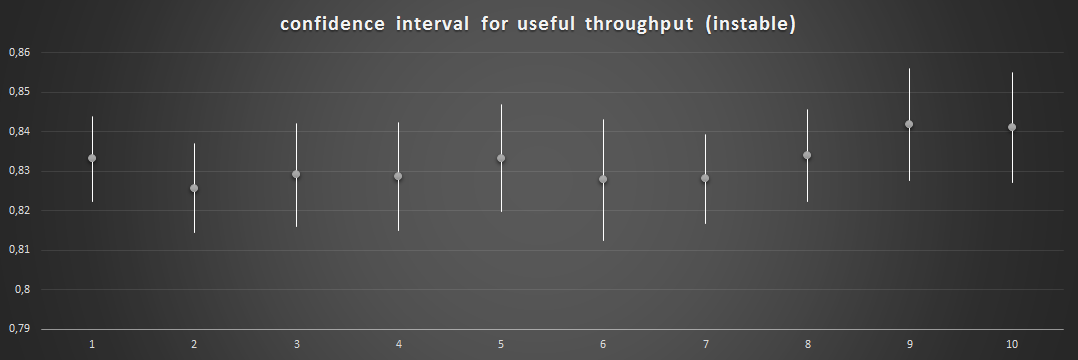
Il throughput del front -end server illustrato nel grafico sopra è stato misurato rispetto alle richieste completate dal front-end per unità di tempo. La convergenza alla stazionarietà in questo caso era previsto dato che il front-end risulta sovraccarico, ed è all’incirca di **218,47** misurato nei run di simulazione di 4000-5000 secondi di simulazione.  
Infine viene riportato di seguito l’istogramma relativo allo useful throughput dato che l’unico indice con comportamento steady-state  
:



*Figura 11: istogramma dello useful throughput*

dal grafico riportato sopra è chiaro come il campione in esame ha una distribuzione di tipo normale con media pari a:

Infine è stato adottato il metodo delle batch means per stimare gli intervalli di confidenza cambiando, per ogni run (fino a 10) gli stream dei generatori random.il grafico seguente illustra i risultati ottenuti stimando 10 intervalli di confidenza con le relative medie:



*Figura 12: istogramma dello useful throughput*

I dettagli relativi alla scelta del metodo batch mean e dei valori di b e k si rimanda nella sezione successiva dell’analisi degli intervalli di confidenza nel caso di sistema con un meccanismo di overload management.

### sistema con OVERLOAD MANAGEMENT

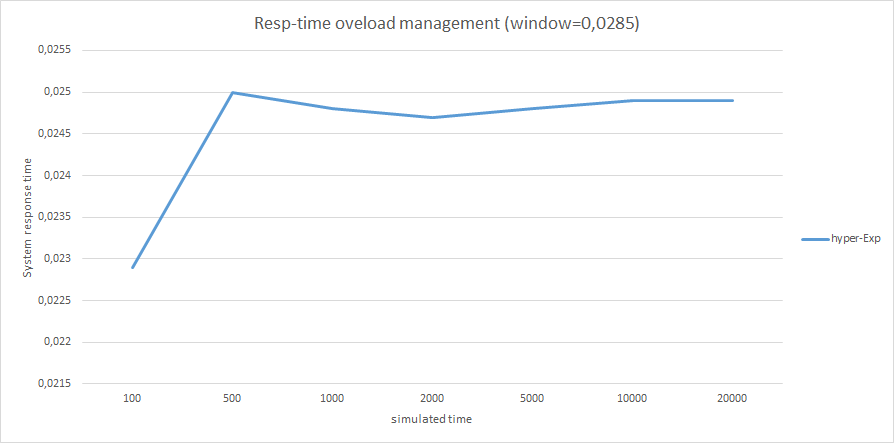
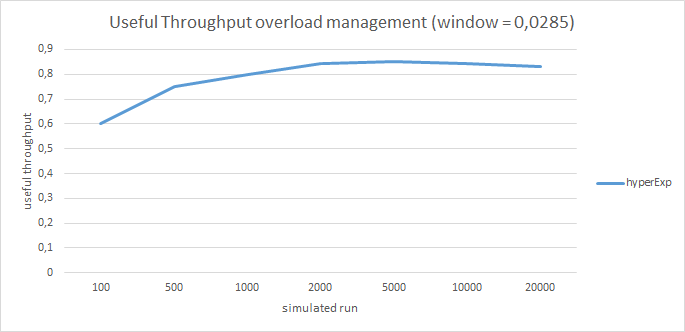
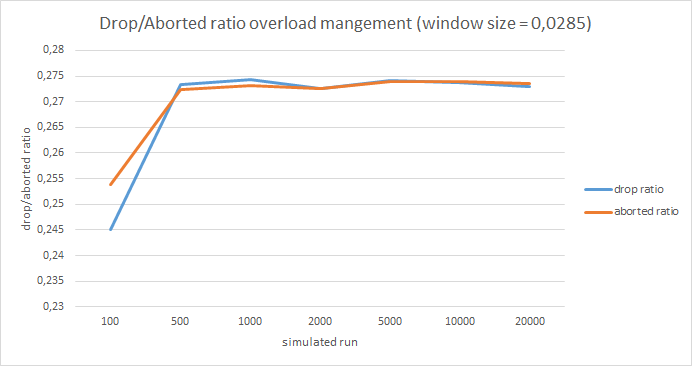
Dai risultati illustrati nel caso di sistema senza il meccanismo di *overload management(*instabilità*)* si è preferito scegliere la distribuzione dei tempi di servizio considerato “peggiore” ossia con i tempi di risposta maggiori rispetto alle altre distribuzioni in esame in modo da integrare nello scenario di applicazione del meccanismo di *overload management*. Tale scelta è ricaduta ovviamente alla distribuzione iperesponenziale.  
Quando viene applicato l’overload management nel front-end server, gli indici di prestazione di interesse assumono un comportamento stazionario per run simulativi abbastanza lunghi.  
Di seguito viene riportato il grafico sul tempo di risposta del sistema con distribuzione dei tempi di servizio iperesponenziale del front-end server:  
  


Figura 13: tempi di risposta del sistema con overload management

Dal grafico riportato sopra si nota come il tempo di risposta del sistema con distribuzione iperesponenziale dei tempi di servizio del front-end server e finestre temporali di osservazione dell’utilizzazione nel front-end impostate a 0,0285 sec assume un valore stazionario di circa **0,025 sec.**Il grafico successivo illustra l’andamento dello useful throughput del sistema con gli parametri sopra citati:



*Figura 14: useful throughput del sistema con overload management*

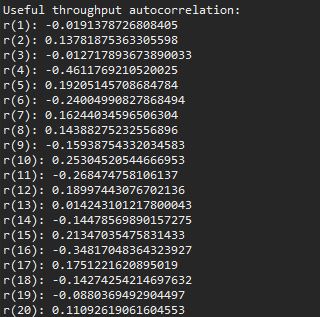
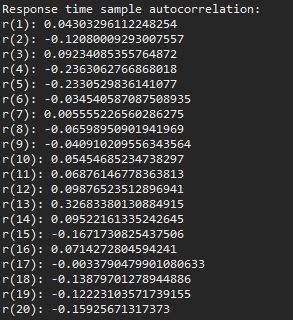
Il valore dello useful throughput in condizioni di stazionarietà assume un valore pari incirca a **0,833** sessioni completate/run.  
Infine vengono riportati gli andamenti degli indici *drop ratio* e *aborted ratio*:  
  


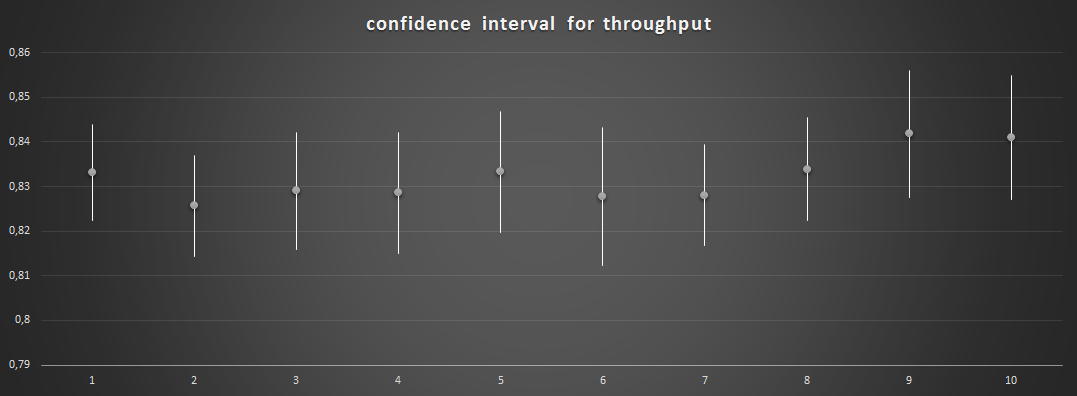
*Figura 15: drop ratio e abort ratio del sistema con overload management*

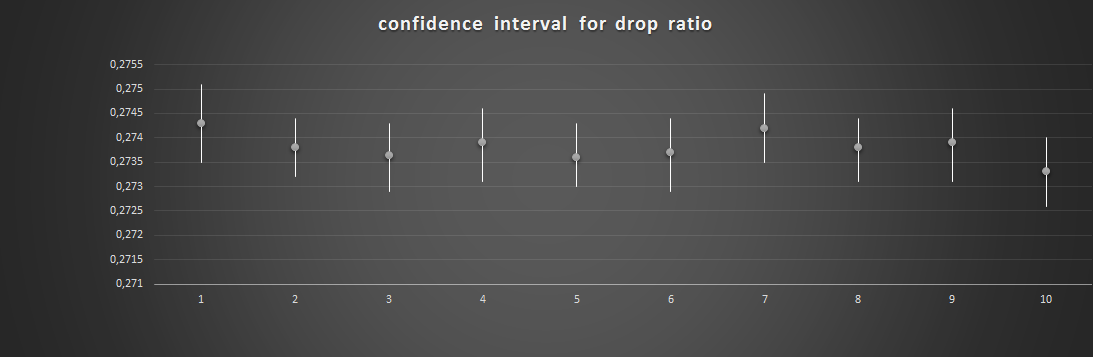
anche tali indici per run di simulazioni lunhi assumono valori stazionari di circa **0,273,** il che significa che il sistema scarta circa il **27%** siadelle nuove sessioni entranti rispetto a quelle totali sia quelle relative alle richieste pendenti.

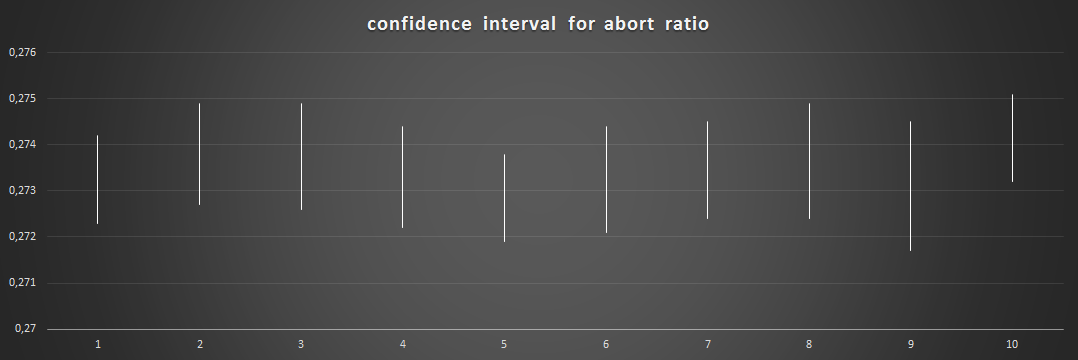
### Batch mean e intervalli di confidenza

Una volta verificate che le statistiche di interesse hanno un comportamento stazionario, si pone l’obiettivo di ottenere delle stime di intervalli per statistiche steady-state, con un livello di confidenza del 95 per cento, il metodo qui adottato è il *metodo delle batch means.*L’idea alla base è quello di effettuare un unico lungo run e partizionare i dati raccolti relative alle statistiche di interesse in partizioni(batch) di uguale lunghezza, calcolare le medie di ciascun batch, e successivamente calcolare la media delle medie dei batch, ricavando anche la deviazione standard relativa per poter costruire gli intervalli di confidenza per la media. Tali intervalli vengono determinati dai rispettivi endpoints:  
]  
  
dove *x* indica la media campionaria (media dei batch) ed *s* la deviazione standard. *t*  è la variabile di Student calcolato usando la funzione[[9]](#footnote-8):  
  
*t =* ***idfStudent(k-1, 1-a/2);***

*a* rappresenta il parametro di confidenza.  
L’indipendenza dei batch è garantita dall’eliminazione della dipendenza dallo stato iniziale del sistema (quando ancora è in fase di stabilizzazione) ottenuta cambiando gli stream(quindi i semi iniziali) dei generatori[[10]](#footnote-9) ad ogni run.Tale cambiamento di stream si traduce in un reset dei valori delle statistiche all’avvio di ogni run di simulazione.  
Particolare attenzione è stata prestata nella scelta dei parametri (**b,k)** necessari per determinare l’ampiezza e il numero dei batch. Seguendo le linee guida del libro[[11]](#footnote-10) si è scelto di impostare un **k=64** fisso mentre b è stato determinato dal rapporto **b=n/k.**Il valore calcolato di b è risultato ottimale anche controllando l’andamento della funzione di autocorrelazione (tendente a zero) sia per i tempi di risposta del sistema che per lo useful throughput:  
Dopo aver impostato il valore di b e k, sono state effettuati 10 run di simulazione, dove per ogni run si settavano dapprima gli stream dei relativi generatori random e al termine del run si calcolano le medie e varianze (utilizzando l’algoritmo di *Welford*).  
I risultati di tali analisi possono essere riassunti dai grafici illustrati di seguito, una per ogni statistica:

  
*Figura 16: intervalli di confidenza per i tempi di risposta*  
  
*Figura 17: intervalli di confidenza per lo useful throughput*

  
*Figura 18: intervalli di confidenza per drop ratio*



*Figura 19: intervalli di confidenza per abort ratio*

Dal momento che sono state eseguite 10 run (quindi ottenuti 10 intervalli di confidenza), si può notare dai grafici qui sopra come il valore medio teorico di ciascuna delle statistiche potrebbe ricadere con una probabilità del 95% in ciascuna degli intervalli, e quindi sui dieci intervalli determinati, 9,5 di questi potrebbero contenere il valore medio teorico. Tali risultati, inoltre, confermano anche la corretta scelta dei parametri di **(b,k)** in quanto, se per esempio si avesse scelto un valore di b molto piccolo (quindi un k molto grande) si potrebbero ottenere dei falsi sensi di confidenza riguardo alla disposizione degli intervalli.

# 

# Conclusioni

In questa seconda parte del progetto è stato effettuato l’analisi delle prestazioni di un sistema che emula uno scenario di traffico web reale.  
Infatti oltre a simulare un comportamento stocastico riguardo sia i tempi di interarrivo delle richieste effettuate dai client che ai loro tempi di processamento nei server e di thinking, fissati i parametri del sistema dalle specifiche riportate e applicando il meccanismo di overload management, il sistema presenta un comportamento simile ai server web reali. Si è analizzato quindi le prestazioni di tale sistema al variare delle distribuzioni dei tempi di servizio, dell applicazione del meccanismo di overload management valutando l’andamento degli indici di prestazione di interesse quali lo useful throughput e i tempi di risposta del sistema, oltre a valutare il drop ratio e l’abort ratio. A tale scopo è stato utilizzato un simulatore next-event il quale integra un meccanismo di avanzamento del tempo simulato basato sull’occorrenza degli eventi schedulati in apposite strutture dati.  
La valutazione degli indici di prestazione del sistema studiato è stata la priorità principale in tutta questa durata di studio, oltre ad essere stato di maggiore interesse per i componenti del gruppo, mentre l’ottimizzazione del codice scritto del simulatore e l’implementazione di una interfacci grafica per un utente sono state considerate di minore priorità per questo progetto, tuttavia potrebbe essere una possibile estensione futura.

# Appendice A

## codice in c della prima parte del progetto

**EXTREMES\_TEST.H**#ifndef \_\_EXTREMES\_TEST\_H\_\_

#define \_\_EXTREMES\_TEST\_H\_\_

struct node

{

double value;

long count;

struct node \*next;

};

typedef struct node \*pointer;

void Init\_list(pointer \*, double);

void Insert(pointer , double);

double Chi\_Stat(double , double);

int Check\_Conf(double, double , double);

void Initialize(void);

void init\_array\_chi\_square(void);

void sort\_chi\_square(void);

void clearScreen(void);

#endif **EXTREMES\_TEST.C**  
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <math.h>

#include "header/rngs.h"

#include "header/rvms.h"

#include "header/testMultiplier.h"

#include "header/extremes\_test.h"

#define K 1000

#define N 10000

#define STREAMS 256

#define DIM 23093

#define D256 0.084

int o[K]; // observed number of times state x occurred

int e[K]; // expected number of times state x will occur

void Init\_list(pointer \*p, double data)

/\* ----------------------------------------

\* initialize a new list node

\* ----------------------------------------

\*/

{

\*p = malloc(sizeof(struct node));

(\*p)->value = data;

(\*p)->count = 1;

(\*p)->next = NULL;

}

void Insert(pointer head, double data)

/\* --------------------------------------

\* add 'data' to the list

\* --------------------------------------

\*/

{

int found = 0;

pointer p; /\* the traveling pointer \*/

pointer q; /\* pointer to the previous node \*/

int temp = 0;

p = head;

while ((!found) && (p != NULL)) { /\* traverse the list until \*/

q = p;

if (p->value == data) /\* either 'data' is found \*/

found = 1; /\* or the list ends \*/

else{

temp = p->count;

p = p->next;

}

}

if (found){

while(p != NULL){

(p->count)++;

p = p->next;

} /\* if 'data' was found \*/

//(p->count)++;

} /\* inc the corresponding count \*/

else { /\* else \*/

Init\_list(&p, data); /\* add a new node (at the end) \*/

q->next = p;

p->count = temp + 1; /\* and link it \*/

}

}

double chi[STREAMS];

static double ALPHA = 0.05;

static long seed = 12345;

void Initialize(void)

{

int x;

int i;

int j;

int d;

double u;

d = 5;

double r;

for(x = 0; x < K; x++)

o[x] = 0;

for(i=0; i < N; i++){

r=Random();

for(j=1; j < d; j++){

u = Random();

if(u > r){

r = u;

}

}

u = exp(d \* log(r));

x = (int)(u \* K);

o[x]++;

}

}

double Chi\_Stat(double v1, double v2)

{

int x;

double sum;

double num;

double den;

num=0.0;

den=0.0;

sum=0.0;

for(x=0; x < K; x++){

num = pow((o[x] - N/K), 2);

den = N/K;

sum += num/den;

}

return sum;

}

int Check\_Conf(double v, double v1, double v2)

{

if (v < v1)

return 1;

else if (v > v2)

return 2;

return 0;

}

void init\_array\_chi\_square(void)

{

int j;

for (j=0; j < STREAMS; j++)

chi[j] = 0.0;

}

void sort\_chi\_square(void)

{

int j;

int c;

int d;

double swap;

for (c = 0 ; c < (STREAMS - 1); c++)

{

for (d = 0 ; d < STREAMS - c - 1; d++)

{

if (chi[d] > chi[d+1])

{

swap = chi[d];

chi[d] = chi[d+1];

chi[d+1] = swap;

}

}

}

}

void clearScreen()

{

const char\* CLEAR\_SCREEN\_ANSI = "\e[1;1H\e[2J";

write(STDOUT\_FILENO,CLEAR\_SCREEN\_ANSI,12);

}

int main(void)

{

int i;

double v;

double v1;

double v2;

int count\_upper;

int count\_lower;

FILE \*file;

pointer head;

char c;

long \*p;

long x;

int k;

long jumpM;

//printf("\033[2J");

clearScreen();

count\_upper=0;

count\_lower=0;

v=0.0;

v1=0.0;

v2=0.0;

printf("\*\*\*\*\*\*\* Extremes Test \*\*\*\*\*\*\*\n");

do{

printf("\nType 1 for generate list of multipliers\n");

printf("Type 2 for check if selected multiplier: %ld, is FULL-PERIOD and MODULECOMPATIBLE\n", sel\_multiplier);

printf("Type 3 for testing the correctness the Random(multiplier)\n");

printf("Type 4 for spectral test!\n");

printf("Type 5 for determinating jump multiplier\n");

printf("Type 6 for execute extremes test and Kolmorov analisys\n");

printf("Type 0 for exit\n");

scanf("%d", &c);

switch (c){

case 1:

printf("create multipliers.txt\n");

FILE \*f = fopen("multipliers.txt", "w");

if(f == NULL )

{

printf("Error opening file!\n");

return -1;

}

p = init\_arrayMultipliers(INIT\_MULTIPLIER);

for(i=0; i < DIM; i++)

{

fprintf(f,"a: %ld\n", \*(p+i));

}

fflush(f);

fclose(f);

printf("done!! multipliers.txt has been created!!\n");

break;

case 2:

if (check(sel\_multiplier))

printf("mulitplier %ld is FULL-PERIOD and MODULE-COMPATIBLE\n", sel\_multiplier);

else

printf("multiplier: %ld is not FULL-PERIOD and/or not MODULE-COMPATIBLE\n", sel\_multiplier);

break;

case 3:

printf("Testing lehmer implementation for multiplier %ld ......\n ", sel\_multiplier);

x = testRandom();

printf("After 10000 calls to testRandom() the value of state is: %ld\n", x);

break;

case 4:

printf("Generating output spectral.dat .... \n");

k = spectralTest();

if(k != -1){

printf("Generating spectral plot ....\n");

system("gnuplot scripts/spectral\_script.txt");

}

else

printf("Error (%d) while generating output spectral.dat!!\n", k);

break;

case 5:

jumpM = genJumpMultiplier(sel\_multiplier);

printf("Jump Multiplier for chosen multiplier: %ld is: %ld\n", sel\_multiplier,

jumpM);

if (check(jumpM))

printf("Jump Multiplier %ld is MODULE-COMPATIBLE with M\n", jumpM);

else

printf("Jump Multiplier %ld in not MODULE-COMPATIBLE with M\n", jumpM);

break;

case 6:

count\_lower = 0;

count\_upper = 0;

init\_array\_chi\_square();

file = fopen("charts/datasets/scatter\_plot.dat", "w");

if(file == NULL){

printf("Error opening file!\n");

return -1;

}

/\* initialize lehmer generator \*/

PlantSeeds(seed);

v1 = idfChisquare(K-1, ALPHA/2);

v2 = idfChisquare(K-1, 1-(ALPHA/2));

fprintf(file,"%d %.3f\n", 0, v1);

fprintf(file,"%d %.3f\n", 255, v1);

fprintf(file,"\n");

fprintf(file, "%d %.3f\n", 0, v2);

fprintf(file, "%d %.3f\n", 255, v2);

fprintf(file, "\n");

for(i=0; i < 256; i++){

SelectStream(i);

Initialize();

v = Chi\_Stat(v1, v2);

chi[i] = v;

fprintf(file,"%d %.3f\n", i, v);

if(Check\_Conf(v,v1,v2) == 1)

count\_lower++;

else if (Check\_Conf(v,v1,v2) == 2)

count\_upper++;

}

fflush(file);

fclose(file);

printf("# test failed in upper bound: %d\n", count\_upper);

printf("# test failed in lower bound: %d\n", count\_lower);

system("gnuplot scripts/script\_scatterplot.txt");

sort\_chi\_square();

Init\_list(&head, chi[0]);

for(i=1; i < STREAMS; i++)

{

Insert(head, chi[i]);

}

file = fopen("charts/datasets/chi\_square\_cdf.dat", "w");

if(file == NULL){

printf("Error opening file!\n");

return -1;

}

pointer k = head;

while(k!= NULL)

{

fprintf(file,"%.5f %.3f\n", k->value, (k->count)/ 256.0);

k = k->next;

}

fflush(file);

fclose(file);

pointer p = head;

FILE \*file1 = fopen("charts/datasets/chi\_cdf.dat", "w");

if(file1 == NULL){

printf("Error opening file!\n");

return -1;

}

double max = 0.0;

double chiMax=0.0;

while(p != NULL)

{

double diff = fabs(cdfChisquare(K - 1 ,p->value) - (p->count)/ 256.0);

fprintf(file1,"%.5f %.3f\n", p->value, cdfChisquare(K - 1 ,p->value));

//printf("cdfChisquare: %.6f\n", cdfChisquare(K - 1, p->value));

//printf("Fn(x): %.6f\n", (p->count) / 256.0);

//printf("diff: %.6f\n", diff);

if(diff > max){

max = diff;

chiMax = p->value;

}

p = p->next;

}

fprintf(file1, "\n%.3f %.3f\n", chiMax, 0.0);

fprintf(file1, "%.3f %.3f\n", chiMax, 1.0);

fflush(file1);

fclose(file1);

printf("d(256) max: %.6f, X(chi-square variable) max is: %.3f\n", max, chiMax);

if(max < D256)

printf("Test of extremes has detected no deficienses in the rngs generator!\n");

else

printf("Test of extremes has detected some deficienses in the rngs generator! RE-CHECK IT PLEASE!\n");

system("gnuplot scripts/chi\_square\_cdf\_script.txt");

break;

case 0:

printf("Exiting....\n");

break;

default:

printf("Invalid input!!\n");

break;

}

if(c <=0 || c> 6){

c = getchar();

ungetc(c, stdin);

c = -1;

}

}while(c > 0 && c <= 6);

return 0;

}

**TESTMULTIPLIER.h**#ifndef \_\_TESTMULTIPLIER\_H\_\_

#define \_\_TESTMULTIPLIER\_H\_\_

extern long sel\_multiplier;

extern long INIT\_MULTIPLIER;

extern long arrayMultipliers[];

int gcd(int, int);

int check(long);

long \*init\_arrayMultipliers(long);

void getSeed(long \*);

long genJumpMultiplier(long);

double RandomGen(long);

double testRandom(void);

int spectralTest(void);

int jumpCheckMC(long);

#endif

**TESTMULTIPLIER.C**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include "header/testMultiplier.h"

#define MODULUS 2147483647L

#define INIT\_SEED 123456789L

#define DIM 23093

long INIT\_MULTIPLIER = 7;

long arrayMultipliers[DIM];

long J = (long)(2147483648u / 256);

long sel\_multiplier = 16807; //50812; //48271; //16807;

static long seed = INIT\_SEED;

int gcd(int a, int b)

{

int c;

while(a != 0){

c = a;

a = b % a;

b = c;

}

return b;

}

long \*init\_arrayMultipliers(long a)

{

long i;

long x;

int index;

long t;

t = seed;

x = a;

index = 0;

i = 1;

while (x != 1){

if ((MODULUS % x < MODULUS / x) && gcd(i, MODULUS - 1) == 1){

/\* x is a full period multiplier equal to a^i mod m \*/

if (index < DIM){

arrayMultipliers[index] = x;

index++;

}

else break;

}

i++;

t = a \* (x % ((long)MODULUS / a)) - (MODULUS % a)\*(x / (long)(MODULUS / a));

if (t > 0){

x = t;

}

else{

x = t + MODULUS;

}

}

return arrayMultipliers;

}

int check(long a)

{

long r;

long q;

long m;

r = MODULUS % a;

q = (long)MODULUS / a;

m = a\*q + r;

if (r < q && m == MODULUS) return 1;

else return 0;

}

void getSeed(long \*x)

{

\*x = seed;

}

long genJumpMultiplier(long a)

{

int j;

long jump;

long t;

long result;

long jmax;

jmax=0;

j = 0;

jump = 1;

t = 1;

result=0;

while(j < J)

{

t = a \* (jump % ((long)MODULUS / a))-(MODULUS % a)\*(jump/(long)(MODULUS / a));

if(t > 0)

jump = t;

else

jump = t + MODULUS;

if(check(jump)){

result = jump;

jmax = j+1;

}

j++;

}

printf("The max jump size which jump multiplier is MC with M is: %ld\n", jmax);

return result;

}

double RandomGen(long multiplier)

{

const long Q = MODULUS / multiplier;

const long R = MODULUS % multiplier;

long t;

t = multiplier \* (seed % Q) - R \* (seed / Q);

if (t > 0)

seed = t;

else

seed = t + MODULUS;

return ((double) seed / MODULUS);

}

double testRandom(void)

{

long x;

long i;

seed = 1L;

printf("\nProgress:\e[s");

for(i = 0; i < 10000; i++){

x = ((float)i / 10000) \* 100;

RandomGen(sel\_multiplier);

printf(" %2ld (%3ld%%)\e[u", i, x);

fflush(stdout);

usleep(1500);

}

getSeed(&x);

return x;

}

int spectralTest(void)

{

double x;

double x1;

long i;

FILE \*file;

seed = INIT\_SEED;

file = fopen("charts/datasets/spectral.dat", "w");

if(file == NULL){

printf("Error opening file!\n");

return -1;

}

x = RandomGen(sel\_multiplier);

for(i=0; i< MODULUS; i++){

x1 = RandomGen(sel\_multiplier);

if(x < 0.001 && x1 < 0.001){

fprintf(file,"%.8f %.8f\n", x, x1);

}

x = x1;

}

fflush(file);

fclose(file);

return 0;

}

int jumpCheckMC(long multiplier)

{

long jumpM;

jumpM = genJumpMultiplier(multiplier);

return check(jumpM);

}

# Appendice B

## codice in java della seconda parte del progetto

## package nextEventSimulator;

## AREA.JAVA

## public class Area {

## private double node;

## private double service;

## private double queue;

## private double istantService;

## private double tLast;

## private double tLastIstant;

## 

## public Area(double START){

## node=service=queue=istantService=0;

## tLast=START;

## tLastIstant=START;

## }

## public double getAreaService(){

## return service;

## }

## public double getIstantAreaService() {

## return istantService;

## }

## public void resetIstantService(double l){

## istantService=0.0;

## tLastIstant=l;

## }

## public void sumAreaService(double service) {

## this.service += service;

## 

## }

## public void sumIstantAreaService(double service){

## this.istantService+=service;

## }

## public double getAreaNode() {

## return node;

## }

## public void sumAreaNode(double node){

## this.node += node;

## }

## public double getAreaQueue(){

## return queue;

## }

## public void sumAreaQueue(double queue) {

## this.queue += queue;

## }

## public double gettLast() {

## return tLast;

## }

## public double gettLastIstant() {

## return tLastIstant;

## }

## public void settLast(double tLast) {

## this.tLast = tLast;

## }

## public void settLastIstant(double tLast) {

## this.tLastIstant = tLast;

## }

## public void reset(double sat){

## node=service=queue=0;

## tLast=sat;

## }

## } CALENDAR.JAVA

## package nextEventSimulator;

## import java.util.HashMap;

## import java.util.Iterator;

## import java.util.Map;

## 

## public class Calendar {

## private Map<Events, Double> calendar;

## 

## public Calendar() {

## calendar = new HashMap<Events, Double>();

## calendar.put(Events.NEW\_SESSION, Main.INFINITY);

## calendar.put(Events.COMPLETION\_FRONT\_SERVER, Main.INFINITY);

## calendar.put(Events.COMPLETION\_BACKEND\_SERVER, Main.INFINITY);

## calendar.put(Events.NEW\_REQUEST, Main.INFINITY);

## calendar.put(Events.MONITORING, Main.INFINITY);

## }

## 

## public void setTimeEvent(Events event, double time) {

## if(time >= 0.0)

## calendar.put(event, time);

## }

## 

## public double getEventTime(Events event){

## return calendar.get(event);

## }

## public Events getNextEvent(){

## Iterator e = calendar.entrySet().iterator();

## double min = Main.INFINITY;

## Events nextEvent = null;

## while(e.hasNext()){

## Map.Entry pair = (Map.Entry)e.next();

## if ((double)pair.getValue() < min){

## min = (double) pair.getValue();

## nextEvent = (Events) pair.getKey();

## }

## }

## return nextEvent;

## }

## }

## CLOCK.JAVA

## package nextEventSimulator;

## 

## public class Clock {

## private double arrival; /\* next arrival time \*/

## private double completion; /\* next completion time \*/

## private double current; /\* current time \*/

## private double next; /\* next (most imminent) event time \*/

## private double last; /\* last arrival time \*/

## 

## public Clock(double start, double completion){

## arrival = 0.0;

## this.completion = completion;

## current = start;

## next = 0.0;

## last = 0.0;

## }

## 

## public double getArrival() {

## return arrival;

## }

## 

## public void setArrival(double arrival) {

## this.arrival = arrival;

## }

## 

## public double getCompletion() {

## return completion;

## }

## 

## public void setCompletion(double completion) {

## this.completion = completion;

## }

## 

## public double getCurrent() {

## return current;

## }

## 

## public void setCurrent(double current) {

## this.current = current;

## }

## 

## public double getNext() {

## return next;

## }

## 

## public void setNext(double next) {

## this.next = next; }}

## EVENTS.JAVA

## package nextEventSimulator;

## public enum Events {

## COMPLETION\_FRONT\_SERVER,

## COMPLETION\_BACKEND\_SERVER,

## NEW\_SESSION,

## MONITORING,

## NEW\_REQUEST

## }

## EVENT SCHEDULER.JAVA

## package nextEventSimulator;

## 

## import java.util.ArrayList;

## import java.util.Collection;

## import java.util.HashMap;

## import java.util.Iterator;

## import java.util.Map;

## import java.util.prefs.BackingStoreException;

## 

## import randomGenerators.Rvgs;

## import randomGenerators.Rvms;

## 

## public class EventScheduler {

## private double INFINITY = Main.INFINITY;

## private Events currentEvent;

## private double stopSim;

## private Clock t;

## private Rvgs rvgs;

## private Rvms rvms;

## private int lengthSessionIndex, interarrivalSessionIndex;

## private double MIN=INFINITY;

## private double MAX = 0.0;

## private boolean reject = false;

## private Statistics stats;

## private ArrayList<Long> sessionLengthSample;

## private int sessionReject = 0;

## private int requestReject = 0;

## private Area system;

## private int completationSessions;

## private ArrayList<Double> responseTimes = new ArrayList<Double>();

## private double avgFrontServiceTime=0;

## private int nAvgFrontServiceTime=0;

## private double avgBackServiceTime=0;

## private double sizeBatch;

## private double resetTime;

## 

## private boolean overloadManagement = false;

## private double sizeMonotoringWindow=0;//0.0285;

## private boolean isBatchMeans=false;

## 

## 

## 

## public EventScheduler(double stop, Clock t){

## this.stopSim = stop;

## this.t = t;

## }

## 

## public void setTimeEvent(Events event, double time, Calendar calendar){

## if(time >= 0.0)

## calendar.setTimeEvent(event, time);

## }

## 

## public void run(Rvgs rvgs, Server frontEnd, Server backEnd, TerminalsServer terminal, Calendar calendar,

## int lengthSessionIndex, int interarrivalSessionIndex, double arrivalMeanTime, int a, int b, int k, double sw, boolean throughputHistogram){

## if(k>1)

## isBatchMeans=true;

## if(sw>0){

## this.sizeMonotoringWindow=sw;

## this.overloadManagement=true;

## System.out.println("Overload management ON");

## }

## 

## this.interarrivalSessionIndex = interarrivalSessionIndex;

## this.lengthSessionIndex = lengthSessionIndex;

## this.rvgs = rvgs;

## this.rvms= new Rvms();

## this.rvgs.rngs.selectStream(this.interarrivalSessionIndex);

## this.system=new Area(Main.START);

## this.completationSessions = 0;

## this.stats = new Statistics();

## this.sessionLengthSample= new ArrayList<Long>();

## ArrayList<Double> utilizationFrontSample = new ArrayList<Double>();

## ArrayList<Double> populationFrontSample = new ArrayList<Double>();

## ArrayList<Double> utilizationBackSample = new ArrayList<Double>();

## ArrayList<Double> populationBackSample = new ArrayList<Double>();

## ArrayList<Double> populationSystemSample = new ArrayList<Double>();

## ArrayList<Double> usefulThroughputSample = new ArrayList<Double>();

## ArrayList<Double> dropRatioSample = new ArrayList<Double>();

## ArrayList<Double> abortedRatioSample = new ArrayList<Double>();

## calendar.setTimeEvent(Events.NEW\_SESSION, rvgs.exponential(arrivalMeanTime)+t.getCurrent());

## int countF=0;

## int countB=0;

## int countS=0;

## int countR=0;

## resetTime=0.0;

## sizeBatch = stopSim/k;

## int i = 1;

## reject = false;

## if(this.overloadManagement && sizeMonotoringWindow>0)

## calendar.setTimeEvent(Events.MONITORING, Main.START+sizeMonotoringWindow);

## 

## while (( t.getCurrent() < stopSim)){

## this.currentEvent = calendar.getNextEvent();

## t.setCurrent(calendar.getEventTime(this.currentEvent)); /\* advance the clock \*/

## 

## 

## if(t.getCurrent()>=i\*sizeBatch){

## System.out.println("End batch: "+i);

## double throughputFront = 0.0;

## double ResponseTimeFront = 0.0;

## double throughputBack = 0.0;

## double ResponseTimeBack = 0.0;

## double ResponseTimeSystem = 0.0;

## 

## utilizationFrontSample.add(frontEnd.getArea().getAreaService()/sizeBatch);

## populationFrontSample.add(frontEnd.getArea().getAreaNode()/sizeBatch);

## utilizationBackSample.add(backEnd.getArea().getAreaService()/sizeBatch);

## populationBackSample.add(backEnd.getArea().getAreaNode()/sizeBatch);

## populationSystemSample.add(system.getAreaNode() / sizeBatch);

## usefulThroughputSample.add(this.completationSessions / sizeBatch);

## abortedRatioSample.add(((double)requestReject)/(countR));

## dropRatioSample.add(((double)sessionReject)/countS);

## throughputFront = countF / sizeBatch;

## ResponseTimeFront = populationFrontSample.get(i-1) / throughputFront;

## throughputBack = countB / sizeBatch;

## ResponseTimeBack = populationBackSample.get(i-1) / throughputBack;

## ResponseTimeSystem = populationSystemSample.get(i-1) / throughputBack;

## 

## if(!isBatchMeans){

## System.out.println(" --> utilizationFront mean: " + utilizationFrontSample.get(i-1));

## System.out.println(" --> populationFront mean: " + populationFrontSample.get(i-1));

## System.out.println(" --> throughputFront mean: " + throughputFront);

## System.out.println(" --> utilizationBack mean: " + utilizationBackSample.get(i-1));

## System.out.println(" --> populationBack mean: " + populationBackSample.get(i-1));

## System.out.println(" --> throughputBack mean: " + throughputBack);

## System.out.println(" --> ResponseTimeBack mean: " + ResponseTimeBack);

## System.out.println(" --> ResponseTime System: " + ResponseTimeSystem );

## System.out.println(" --> populationSystem mean: " + populationSystemSample.get(i-1));

## System.out.println(" --> usefulThroughput: " + usefulThroughputSample.get(i-1));

## System.out.println(" --> drop ratio: " + (dropRatioSample.get(i-1)));

## System.out.println(" --> aborted ratio: " + (abortedRatioSample.get(i-1)));

## }

## 

## 

## sessionReject = 0;

## requestReject = 0;

## countS=0;

## countR=0;

## frontEnd.getArea().reset(i\*sizeBatch);

## backEnd.getArea().reset(i\*sizeBatch);

## system.reset(i\*sizeBatch);

## this.completationSessions = 0;

## countB=0;

## i++;

## }

## 

## if (this.currentEvent.equals(Events.NEW\_SESSION)){

## countS++;

## newSession(terminal, frontEnd, arrivalMeanTime, a, b, calendar,backEnd);

## }

## else if (this.currentEvent.equals(Events.NEW\_REQUEST)){

## countR++;

## newRequest(terminal, frontEnd, calendar, backEnd);

## }

## else if (this.currentEvent.equals(Events.COMPLETION\_BACKEND\_SERVER)){

## countB++;

## completeBackEnd(terminal, backEnd, calendar, frontEnd);

## }

## else if (this.currentEvent.equals(Events.COMPLETION\_FRONT\_SERVER)){

## countF++;

## completeFrontEnd(frontEnd,backEnd, calendar);

## }

## else if(this.currentEvent.equals(Events.MONITORING)){

## 

## monitoringUtilization(frontEnd, calendar);

## }

## }

## if(isBatchMeans)

## {

## this.batchMeans(this.responseTimes,usefulThroughputSample, abortedRatioSample,dropRatioSample, k,0.95,true);

## if(throughputHistogram)

## stats.createHistogram(usefulThroughputSample, 7,Main.INFINITY);

## 

## }

## 

## }

## 

## 

## 

## private void monitoringUtilization(Server frontEnd, Calendar calendar) {

## calendar.setTimeEvent(Events.MONITORING, t.getCurrent() + sizeMonotoringWindow);

## double utilization= frontEnd.getArea().getIstantAreaService() / (t.getCurrent() - resetTime);

## resetTime=t.getCurrent();

## frontEnd.getArea().resetIstantService(t.getCurrent());

## if(reject && utilization < 0.75){

## reject = false;

## }

## if(!reject && utilization > 0.85){

## reject = true;

## }

## }

## 

## private void batchMeans(ArrayList<Double> responseTimeSample, ArrayList<Double> throughputSample, ArrayList<Double> abortedSample, ArrayList<Double> dropSample, int k,double confidenceLevel, boolean isJobAverage) {

## 

## int n=responseTimeSample.size();

## int r=n%k;

## if(r>0)

## {

## responseTimeSample.subList(n-r, n).clear();

## n=responseTimeSample.size();

## }

## 

## int b=n/k;

## System.out.println("batchMeans b="+b+" n="+n+ " k="+k);

## 

## ArrayList<Double> sampleMeansList=new ArrayList<Double>();

## 

## ArrayList<Double> result;

## 

## ArrayList<Double> temp=new ArrayList<Double>();

## 

## for(int i=0;i<k;i++){

## for(int j=b\*i;j<i\*b+b;j++)

## temp.add(responseTimes.get(j));

## 

## result=stats.MeanVarWelford(temp);

## sampleMeansList.add(result.get(0));

## temp.clear();

## }

## 

## result=stats.MeanVarWelford(sampleMeansList);

## System.out.println("Response time sample autocorrelation:");

## for(int z=1; z <= 20; z++){

## System.out.println("r("+z+"): " + stats.autocorrelationSample(z, throughputSample));

## }

## 

## 

## System.out.println("RESPONSE TIME");

## stats.confidenceIterval(sampleMeansList, confidenceLevel,rvms);

## System.out.println("\nTHROUGHPUT");

## stats.confidenceIterval(throughputSample, confidenceLevel,rvms);

## System.out.println("\nDROPO RATIO");

## stats.confidenceIterval(dropSample, confidenceLevel,rvms);

## System.out.println("\nABORTED RATIO");

## stats.confidenceIterval(abortedSample, confidenceLevel,rvms);

## 

## }

## 

## 

## public void newSession(TerminalsServer term, Server front, double arrivalMeanTime,int a, int b, Calendar calendar, Server backend){

## 

## Area frontArea = front.getArea();

## rvgs.rngs.selectStream(this.interarrivalSessionIndex);

## calendar.setTimeEvent(Events.NEW\_SESSION, calendar.getEventTime(Events.NEW\_SESSION) + rvgs.exponential(arrivalMeanTime));

## if(!reject){

## rvgs.rngs.selectStream(this.lengthSessionIndex);

## long numRequests = rvgs.equilikely(a, b);

## sessionLengthSample.add(numRequests);

## Session session = new Session(numRequests, t.getCurrent());

## term.addTerminal(session);

## frontArea.sumAreaNode((t.getCurrent() - frontArea.gettLast()) \* front.getCountjobs());

## system.sumAreaNode((t.getCurrent() - system.gettLast()) \* (front.getCountjobs()+backend.getCountjobs()));

## Job newJob = session.createJob(t.getCurrent());

## newJob.setRespStartTime(t.getCurrent());

## double tc = front.send(newJob);

## if (tc >= 0){

## front.addBusyTime(tc);

## avgFrontServiceTime+=tc;

## nAvgFrontServiceTime++;

## calendar.setTimeEvent(Events.COMPLETION\_FRONT\_SERVER, tc + t.getCurrent());

## }

## else{

## frontArea.sumAreaService(t.getCurrent() - frontArea.gettLast());

## frontArea.sumIstantAreaService(t.getCurrent() - frontArea.gettLastIstant());

## }

## frontArea.settLast(t.getCurrent());

## frontArea.settLastIstant(t.getCurrent());

## system.settLast(t.getCurrent());

## }

## else{

## sessionReject++;

## }

## 

## }

## 

## public void newRequest(TerminalsServer term, Server front, Calendar calendar, Server backend){

## Area frontArea = front.getArea();

## Job minJob = term.getminJob();

## minJob.setThinking(false);

## term.setMinThinkingjob(term.getMinThinkingJob());

## double time = term.getminJob().getStartTime();

## calendar.setTimeEvent(Events.NEW\_REQUEST, time);

## if(!reject){

## frontArea.sumAreaNode((t.getCurrent() - frontArea.gettLast()) \* front.getCountjobs());

## system.sumAreaNode((t.getCurrent() - system.gettLast()) \*( front.getCountjobs()+backend.getCountjobs()));

## 

## minJob.setRespStartTime(t.getCurrent());

## 

## double tc = front.send(minJob);

## if(tc >= 0){

## front.addBusyTime(tc);

## avgFrontServiceTime+=tc;

## nAvgFrontServiceTime++;

## calendar.setTimeEvent(Events.COMPLETION\_FRONT\_SERVER, tc + t.getCurrent());

## }

## else{

## frontArea.sumAreaService(t.getCurrent() - frontArea.gettLast());

## frontArea.sumIstantAreaService(t.getCurrent() - frontArea.gettLastIstant());

## }

## frontArea.settLast(t.getCurrent());

## frontArea.settLastIstant(t.getCurrent());

## system.settLast(t.getCurrent());

## }

## else{

## requestReject++;

## term.removeSession(minJob.getSession\_id());

## }

## 

## }

## 

## public void completeFrontEnd(Server front, Server back, Calendar calendar){

## Area frontArea = front.getArea();

## Area backArea = back.getArea();

## Job job = front.getServiceJob();

## frontArea.sumAreaNode((t.getCurrent() - frontArea.gettLast()) \* front.getCountjobs());

## 

## 

## double tc = front.executeNextJob();

## 

## if (tc >= 0){

## front.addBusyTime(tc);

## avgFrontServiceTime+=tc;

## nAvgFrontServiceTime++;

## calendar.setTimeEvent(Events.COMPLETION\_FRONT\_SERVER, tc + t.getCurrent());

## }

## else{

## calendar.setTimeEvent(Events.COMPLETION\_FRONT\_SERVER, INFINITY);

## }

## 

## frontArea.sumAreaService(t.getCurrent() - frontArea.gettLast());

## frontArea.sumIstantAreaService(t.getCurrent() - frontArea.gettLastIstant());

## backArea.sumAreaNode((t.getCurrent() - backArea.gettLast()) \* back.getCountjobs());

## tc = back.send(job);

## 

## if (tc >= 0){

## 

## calendar.setTimeEvent(Events.COMPLETION\_BACKEND\_SERVER, tc + t.getCurrent());

## }

## else{

## backArea.sumAreaService(t.getCurrent() - backArea.gettLast());

## }

## 

## frontArea.settLast(t.getCurrent());

## frontArea.settLastIstant(t.getCurrent());

## backArea.settLast(t.getCurrent());

## }

## 

## public void completeBackEnd(TerminalsServer term, Server back, Calendar calendar, Server front){

## Area backArea = back.getArea();

## Job job = back.getServiceJob();

## backArea.sumAreaNode((t.getCurrent() - backArea.gettLast()) \* back.getCountjobs());

## system.sumAreaNode((t.getCurrent() - system.gettLast()) \* (back.getCountjobs()+front.getCountjobs()));

## double tc = back.executeNextJob();

## 

## if (tc >= 0){

## 

## calendar.setTimeEvent(Events.COMPLETION\_BACKEND\_SERVER, tc + t.getCurrent());

## }

## else{

## calendar.setTimeEvent(Events.COMPLETION\_BACKEND\_SERVER, INFINITY);

## }

## backArea.sumAreaService(t.getCurrent() - backArea.gettLast());

## backArea.settLast(t.getCurrent());

## system.settLast(t.getCurrent());

## 

## int sessionJob\_id = job.getSession\_id();

## term.decJobPerSession(sessionJob\_id);

## 

## job.setRespStopTime(t.getCurrent());

## responseTimes.add(job.getRespStopTime() - job.getRespStartTime());

## if(job.getRespStopTime() - job.getRespStartTime() < MIN)

## MIN = job.getRespStopTime() - job.getRespStartTime();

## if(job.getRespStopTime() - job.getRespStartTime() > MAX)

## MAX = job.getRespStopTime() - job.getRespStartTime();

## 

## if (term.isLastJobPerSession(sessionJob\_id)){

## term.removeSession(sessionJob\_id);

## this.completationSessions++;

## }

## else{

## double minTimeCreatejob= term.send(job, t.getCurrent());

## calendar.setTimeEvent(Events.NEW\_REQUEST, minTimeCreatejob);

## }

## }

## }

## FIFO.JAVA

## package nextEventSimulator;

## 

## import java.util.ArrayList;

## 

## public class FIFO extends Queue{

## 

## private ArrayList<Job> jobQueue; //job queue

## 

## public FIFO() {

## jobQueue = new ArrayList<Job>();

## super.setType("FIFO");

## }

## 

## public void insert(Job j) {

## jobQueue.add(j);

## }

## 

## public Job getNext() {

## if(jobQueue.size() > 0)

## return jobQueue.remove(0);

## else

## return null;

## }

## 

## public int getSize() {

## return jobQueue.size();

## }

## 

## public void reset() {

## jobQueue.clear();

## }

## }

## JOB.JAVA

## package nextEventSimulator;

## 

## public class Job {

## private int id;

## private int session\_id;

## private static int count = 0;

## private double startTime;

## private double stopRespTime;

## private double startRespTime;

## private boolean isThinking;

## 

## public Job(double thinkingTime){

## this.startTime = thinkingTime;

## }

## public Job(int sessionId){

## id = count;

## count++;

## startRespTime=0.0;

## stopRespTime=0.0;

## this.session\_id = sessionId;

## setStartJobTime(0.0);

## this.setThinking(false);

## }

## public int getSession\_id() {

## return session\_id;

## }

## public int getId() {

## return id;

## }

## public void setId(int id) {

## this.id = id;

## }

## public double getStartTime() {

## return startTime;

## }

## public void setStartJobTime(double startTime) {

## this.startTime = startTime;

## }

## public boolean isThinking() {

## return isThinking;

## }

## 

## public void setThinking(boolean isThinking) {

## this.isThinking = isThinking;

## }

## public void setRespStartTime(double startRespTime){

## this.startRespTime = startRespTime;

## }

## public void setRespStopTime(double stopRespTime){

## this.stopRespTime = stopRespTime;

## }

## public double getRespStartTime(){

## return this.startRespTime;

## }

## 

## public double getRespStopTime(){

## return this.stopRespTime;

## }

## }

## MAIN.JAVA

## package nextEventSimulator;

## 

## import java.util.Scanner;

## 

## import randomGenerators.\*;

## 

## public class Main {

## 

## 

## private final double arrivalSessionMeanTime = 1.0 / 35;

## private final double meanServiceTimeFrontServer = 0.00456;

## private final double meanServiceTimeBackEndServer = 0.00117;

## private final double meanThinkingTime = 7.0;

## 

## private final int a = 5;

## private final int b = 35;

## 

## 

## private Clock clock;

## private Calendar calendar;

## private EventScheduler scheduler;

## private Server frontServer, backEndServer;

## private TerminalsServer terminalServer;

## private int streamSessionLenght=0;

## private int streamFront=1;

## private int streamBack=2;

## private int streamTerm=3;

## private int streamArrivalSession=4;

## 

## private int shiftStream=10;

## public final static double START = 0.0; /\* initial time \*/

## public static double STOP = 2000; //0.09376; /\* terminal time \*/

## private final static Rvgs rvgs = new Rvgs();

## public final static double INFINITY = (100.0 \* STOP); /\*must be much larger than STOP \*/

## private double sizeMonitoringWindow=0;//0.0285

## private int k = 64;

## private boolean throughputHistogram=false;

## 

## 

## 

## public Main(){

## this.shiftStream(shiftStream);

## clock = new Clock(START, INFINITY);

## frontServer = new Server(1, 1.0/meanServiceTimeFrontServer, rvgs, streamFront, Distributions.EXPONENTIAL);

## backEndServer = new Server(2, 1.0/meanServiceTimeBackEndServer, rvgs, streamBack, Distributions.EXPONENTIAL);

## terminalServer = new TerminalsServer(meanThinkingTime, rvgs, streamTerm);

## calendar = new Calendar();

## scheduler = new EventScheduler(STOP, clock);

## }

## 

## public void shiftStream(int n){

## int shift=(n\*5)%256;

## streamSessionLenght=(streamSessionLenght+shift);

## streamFront=(streamFront+shift);

## streamBack=(streamBack+shift);

## streamTerm=(streamTerm+shift);

## streamArrivalSession=(streamArrivalSession+shift);

## 

## }

## 

## public static void main(String[] args) {

## Main nextSim = new Main();

## nextSim.run();

## 

## }

## 

## private void run() {

## Scanner scan = new Scanner(System.in);

## menuList();

## boolean first=false;

## int c=-1;

## while(c!=0){

## if(first){

## System.out.println("");

## menuList();

## }

## c = scan.nextInt();

## switch(c){

## case 1:

## System.out.print("Select number of batch means(K=1 no batch mean): ");

## k = scan.nextInt();

## if(k <0)

## System.out.print("Error: K must be > 0");

## System.out.print("number K of batch means: " + k);

## break;

## case 2:

## System.out.print("type '1' for throughput histogram or '0' otherwise: ");

## int e = scan.nextInt();

## if(e == 1)

## throughputHistogram = true;

## else if (e==0)

## throughputHistogram = false;

## else

## System.out.print("Error in typing");

## break;

## case 3:

## System.out.print("insert size window(integer >=0): ");

## double i = scan.nextDouble();

## if(i < 0)

## System.out.print("Error: size must be >=0");

## else

## sizeMonitoringWindow =i;

## 

## break;

## case 4:

## System.out.print("insert a positive integer: STOP --> ");

## int f = scan.nextInt();

## if(f<=0)

## System.out.print("Error: STOP must be > 0 ");

## else

## STOP=f;

## break;

## case 5:

## System.out.println("STOP time: " + STOP);

## System.out.println("Number of batch mean: " + k);

## if(!throughputHistogram)

## System.out.println("Activate histogram throughput: " + "Not active");

## else

## System.out.println("Activate histogram throughput: " + "Active");

## System.out.println("Size utilization window monitoring: " + sizeMonitoringWindow);

## 

## break;

## case 6:

## System.out.println("next-event Simulator started!");

## scheduler.run(rvgs, frontServer, backEndServer, terminalServer, calendar, streamSessionLenght, streamArrivalSession, arrivalSessionMeanTime, a, b, k, sizeMonitoringWindow, throughputHistogram);

## 

## break;

## case 0:

## System.out.println("Exit");

## break;

## }

## System.out.println("");

## if(c == 0)

## break;

## else

## first=true;

## 

## }

## scan.close();

## 

## }

## public void menuList(){

## System.out.println("Type 1 for setting batch means");

## System.out.println("Type 2 for computing throughput histogram");

## System.out.println("Type 3 for setting size utilization window monitoring: ");

## System.out.println("Type 4 for setting stop time simulator: ");

## System.out.println("Type 5 for view simulator's parameters");

## System.out.println("Type 6 for start simulator!");

## System.out.println("Type 0 for exit");

## System.out.print("\nSelect type: ");

## }

## 

## }

## QUEUE.JAVA

## package nextEventSimulator;

## 

## import java.io.Serializable;

## 

## public abstract class Queue implements Serializable{

## private String type;

## 

## public void setType(String t)

## {

## type=t;

## }

## public String getType()

## {

## return type;

## }

## public abstract Job getNext();

## 

## public abstract void insert(Job j);

## 

## public abstract int getSize();

## 

## public abstract void reset();

## 

## 

## }

## SERVER.JAVA

## package nextEventSimulator;

## 

## import randomGenerators.Rvgs;

## import randomGenerators.Distributions;

## 

## public class Server {

## private int id;

## private double rateOfService; /\* rate of service \*/

## private long countJobs;

## private FIFO fifoQueue;

## private Distributions dis;

## private Rvgs rvgs;

## private Job serviceJob;

## private int indexStream;

## private Area area;

## private final int K=10;

## private final double P=0.1;

## private double busyTime=0;

## private int count=0;

## 

## public Server(int id, double rateService, Rvgs rvgs, int index, randomGenerators.Distributions dis){

## this.rvgs = rvgs;

## this.indexStream = index;

## this.dis=dis;

## this.id = id;

## this.rateOfService = rateService;

## countJobs = 0;

## fifoQueue = new FIFO();

## area = new Area(Main.START);

## }

## 

## public double getService(){

## this.rvgs.rngs.selectStream(this.indexStream);

## if(this.dis.equals(randomGenerators.Distributions.EXPONENTIAL))

## return this.rvgs.exponential(1.0 / rateOfService);

## else

## if(this.dis.equals(randomGenerators.Distributions.ERLANG))

## {

## return this.rvgs.erlang(K, 1.0 / rateOfService);

## }

## else

## if(this.dis.equals(randomGenerators.Distributions.HYPEREXPONENTIAL))

## return this.rvgs.hyperExponential(P, 1.0 / rateOfService);

## else {

## System.out.println("-1 distribution");

## return -1;

## }

## 

## }

## 

## public double send(Job job){

## countJobs++;

## if(countJobs > 1){

## fifoQueue.insert(job);

## return -1.0;

## }

## else {

## this.serviceJob = job;

## return getService();

## }

## }

## 

## public Area getArea(){

## return this.area;

## }

## 

## public double executeNextJob(){

## countJobs--;

## 

## if(fifoQueue.getSize() > 0){

## this.serviceJob = fifoQueue.getNext();

## return getService();

## }

## else{

## this.serviceJob = null;

## return -1.0;

## }

## }

## 

## public long getCountjobs(){

## return this.countJobs;

## }

## 

## public Job getServiceJob(){

## return this.serviceJob;

## }

## 

## public int getId(){

## return this.id;

## }

## 

## public double getBusyTime(){

## return this.busyTime;

## }

## public void addBusyTime(double t){

## busyTime+=t;

## count++;

## }

## 

## public int getCount(){

## return count;

## }

## 

## 

## }

## SESSION.JAVA

## package nextEventSimulator;

## 

## public class Session {

## private long numJobs;

## private int id;

## private static int countSession=0;

## private Job job;

## 

## public Session(long numRequests, double startTime){

## id = countSession;

## countSession++;

## this.numJobs = numRequests;

## job= null;

## }

## 

## public long geNumJobs() {

## return numJobs;

## }

## 

## public Job createJob(double currentTime){

## this.job = new Job(id);

## this.job.setStartJobTime(currentTime);

## return this.job;

## }

## 

## public void decNumJobs(){

## this.numJobs--;

## }

## 

## public int getId(){

## return this.id;

## }

## 

## public Job getJob(){

## return this.job;

## }

## 

## public void setJob(Job job){

## this.job = job;

## }

## 

## public Job deleteJob(){

## Job g = this.job;

## this.job = null;

## return g;

## }

## 

## public double getStartTimeSessionJob(){

## return this.job.getStartTime();

## }

## 

## public int getCountSession(){

## return this.countSession;

## }

## 

## }

## STATISTIC.JAVA

## package nextEventSimulator;

## 

## import java.util.ArrayList;

## 

## import randomGenerators.Rvms;

## 

## public class Statistics {

## private ArrayList<Double> meanVar;

## 

## public Statistics(){

## meanVar = new ArrayList<Double>();

## }

## 

## public ArrayList<Double> MeanVarWelford(ArrayList<Double> values){

## double mean = 0.0;

## double variance = 0.0;

## ArrayList<Double> results = new ArrayList<Double>();

## for(int i=1; i <= values.size(); i++){

## variance = variance + (((double)(i-1)) / i)\*Math.pow((values.get(i-1) - mean),2);

## mean = mean + (1.0/i)\*(values.get(i-1) - mean);

## }

## variance = variance / values.size();

## results.add(mean);

## results.add(variance);

## this.meanVar = results;

## return results;

## }

## 

## public ArrayList<Double> getmeanVarWelford(){

## return this.meanVar;

## }

## 

## public double autocorrelationSample(int j, ArrayList<Double> sample){

## double r = 0.0;

## double c = 0.0;

## double sum = 0.0;

## 

## double mean = 0.0;

## double variance = 0.0;

## ArrayList<Double> meanVar = MeanVarWelford(sample);

## mean = meanVar.get(0);

## variance = meanVar.get(1);

## for(int i=1; i <= sample.size() - j; i++){

## sum += (sample.get(i-1) - mean)\*(sample.get(i-1+j)- mean);

## }

## c = (1.0 / (sample.size() - j))\* sum;

## r = c / variance;

## return r;

## }

## 

## public void confidenceIterval(ArrayList<Double> meanSample,double confidenceLevel, Rvms rvms){

## double alfa=1-confidenceLevel;

## int k=meanSample.size();

## double t=rvms.idfStudent(k-1, 1-alfa/2);

## double standardDevation=0;

## double mean=0;

## ArrayList<Double> result=this.MeanVarWelford(meanSample);

## mean=result.get(0);

## standardDevation=Math.sqrt(result.get(1));

## double left=mean-(t\*standardDevation)/Math.sqrt(k-1);

## double right=mean+(t\*standardDevation)/Math.sqrt(k-1);

## System.out.println("confidence interval: "+ (confidenceLevel\*100) +"% ["+left+", "+right+"]");

## System.out.println("mid point: "+ mean+ "\n delta:= "+(right-left));

## 

## }

## 

## public void createHistogram(ArrayList<Double> sample,int k, double INFINITY){

## double MIN = INFINITY;

## double MAX = 0.0;

## for(int i=0; i<sample.size(); i++){

## if(sample.get(i) < MIN)

## MIN = sample.get(i);

## if(sample.get(i) > MAX)

## {

## MAX = sample.get(i);

## }

## }

## 

## System.out.println("THROUGHPUT HISTOGRAM");

## System.out.println("MIN: " + MIN + " MAX: " + MAX);

## System.out.println("sample size:"+sample.size());

## 

## int j = 0; /\* histogram bin index \*/

## double DELTA = ((MAX-MIN) / k);

## long outliers\_low = 0;

## long outliers\_high = 0;

## long count[] = new long[k];; /\* bin count \*/

## double midpoint[] = new double[k]; /\* bin midpoint \*/

## double mean = 0.0;

## 

## 

## for (j = 0; j < k; j++){

## count[j] = 0;

## midpoint[j] = MIN + (j + 0.5) \* DELTA;

## 

## }

## 

## System.out.println("delta"+DELTA);

## 

## for(int z =0; z < sample.size(); z++){

## if ((sample.get(z) >= MIN) && (sample.get(z) < MAX)) {

## j = (int)(((sample.get(z) - MIN) / DELTA));

## count[j]++;

## }

## else if (sample.get(z) < MIN)

## outliers\_low++;

## else

## outliers\_high++;

## }

## double sum=0;

## System.out.println("Bins");

## for(int i=0;i<k;i++){

## sum+=count[i]\*midpoint[i];

## System.out.println(count[i]);

## 

## }

## mean=sum/sample.size();

## System.out.println("--> histogram mean: " + mean);

## }

## 

## 

## 

## }

## TERMINALSERVER.JAVA

## package nextEventSimulator;

## 

## import java.util.ArrayList;

## 

## import randomGenerators.Rvgs;

## 

## public class TerminalsServer {

## private ArrayList<Session> N;

## private double meanThinkingTime;

## private Rvgs rvgs;

## private Job minJob;

## private final int indexStream;

## private Area area;

## 

## public TerminalsServer(double z, Rvgs rvgs, int index){

## N = new ArrayList<Session>();

## this.rvgs = rvgs;

## this.indexStream= index;

## meanThinkingTime = z;

## this.minJob = null;

## this.area= new Area(Main.START);

## }

## 

## public int getTerminal() {

## return N.size();

## }

## 

## public void addTerminal(Session n) {

## N.add(n);

## }

## 

## public Area getArea(){

## return this.area;

## }

## 

## public Session removeSession(int sessionId){

## for(int i=0; i < this.N.size(); i++)

## if(this.N.get(i).getId() == sessionId){

## //System.out.println("remove"+sessionId);

## return this.N.remove(i);

## }

## return null;

## }

## 

## public double getThinkingTime() {

## this.rvgs.rngs.selectStream(indexStream);

## return this.rvgs.exponential(meanThinkingTime);

## }

## 

## public Session getSessionById(int id){

## for(int i=0; i < this.N.size(); i++)

## if(this.N.get(i).getId() == id){

## return this.N.get(i);

## }

## return null;

## }

## 

## public void decJobPerSession(int sessionId){

## Session session = getSessionById(sessionId);

## session.decNumJobs();

## }

## 

## public boolean isLastJobPerSession(int sessionId){

## Session session = getSessionById(sessionId);

## if(session.geNumJobs() == 0){

## return true;

## }

## return false;

## }

## 

## public double send(Job job, double currentTime){

## int session\_id = job.getSession\_id();

## Session session = getSessionById(session\_id);

## double thinkingTime = getThinkingTime();

## Job newJob = session.createJob(currentTime);

## newJob.setStartJobTime(currentTime + thinkingTime);

## newJob.setThinking(true);

## session.setJob(newJob);

## this.minJob = getMinThinkingJob();

## return this.minJob.getStartTime();

## 

## }

## 

## public Job getMinThinkingJob(){

## Job minJob = new Job(Main.INFINITY);

## for(int i=0; i < this.N.size(); i++){

## Job job = this.N.get(i).getJob();

## if(job.isThinking()){

## if(job.getStartTime() < minJob.getStartTime()){

## minJob = job;

## }

## }

## }

## return minJob;

## }

## 

## public void setMinThinkingjob(Job job){

## this.minJob = job;

## }

## 

## public Job getminJob(){

## return this.minJob;

## }

## 

## public int getSizeActiveSessions(){

## return this.N.size();

## }

## 

## }

## 

## 

## 

1. *Discrete-Event Simulation: A first course* (Lawrence lewis, Steve Park) [↑](#footnote-ref-0)
2. Suggerito da Lewis, Goodman e Miller(1969) [↑](#footnote-ref-1)
3. Si ricorda che il moltiplicatore full-period e module-compatible con m scelto è: *a=50812* [↑](#footnote-ref-2)
4. Introdotto da Knuth (1998) [↑](#footnote-ref-3)
5. il valore di j usato da rngs.c che garantiva che la funzione di salto era module-compatible con m era j= 8367782, riportato a pag 114 del libro: *Discrete-Event Simulation: a first course* [↑](#footnote-ref-4)
6. Teorema 10.1.1 pag. 442 del libro Discrete Event simulation: a first course [↑](#footnote-ref-5)
7. Nominato anche come *Two-tailed test* consigliato dal libro e da Knuth (pag.441) [↑](#footnote-ref-6)
8. Noto come il teorema di Glivenko-Cantelli. *Elementi di Statistica inferenziale - Barbara Torti* [↑](#footnote-ref-7)
9. fornito dalla libreria *rvms.c* [↑](#footnote-ref-8)
10. forniti dalla libreria *rngs.c* [↑](#footnote-ref-9)
11. capitolo 8 pag 390, per la scelta di (b,k) [↑](#footnote-ref-10)