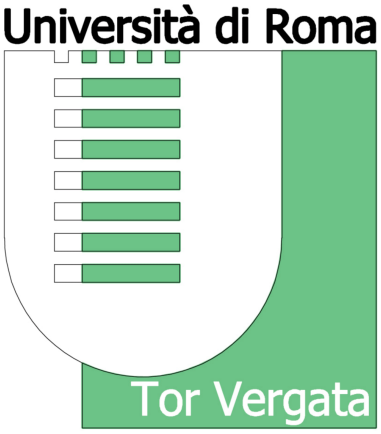
Modelli di Prestazioni di Sistemi e Reti

Simulatore di traffico in un sistema "multi-tier"

Simone Martucci - simone.martucci.91@gmail.com
Alessandro Valenti - alessandro.valenti1991@gmail.com



Indice

Capitolo 1

Generatore di Lehmer

Il progetto si pone di testare il noto generatore pseudo-casuale di numeri random di Lehmer utilizzando, a tale scopo, uno dei test di casualitá illustrati nel libro *Discrete-Event Simulation: A first course*. All'interno del progetto si é scelto di utilizzare la versione del generatore fornita come libreria C dallo stesso libro, é stato quindi creato un programma C che si interfaccia con tali librerie, tramite chiamate alle API della libreria stessa, che serva a testare la effettiva correttezza di tale implementazione. Al fine di comprendere al meglio i risultati ottenuti verranno anche presentati dei grafici riassuntivi del test effettuato.

1.1 Funzionamento

Il *generatore di Lehmer* é un generatore basato su un algoritmo che da origine ad una sequenza di numeri pseudo-casuali. É definito da due parametri:

- un modulo $\boldsymbol{m}$, che é un numero primo molto grande (in questo caso la libreria usa $2^{31} - 1$);
- un moltiplicatore $\boldsymbol{a}$ che rappresenta un numero intero compreso tra 1 ed $m - 1$.

La sequenza numerica pseudo-random viene generata tramite la formula :

$$x_{i+1} = ax_i mod m$$

Questa sequenza parte da un numero x_0 detto seed, anch'esso scelto tra 1 ed $m - 1$. Non tutte le combinazioni di a ed m peró sono ottimali per

realizzare una sequenza di numeri che garantiscano una buona randomicità. Per verificare dunque se un seed e un moltiplicatore garantiscono un buon livello di randomicità esistono dei test empirici, nel capitolo successivo tale generatore verrà sottoposto ad uno di questi.

1.2 Test degli estremi

Il test scelto per effettuare la verifica sul generatore, con parametri:

$$(a, m) = (48271, 2^{31} - 1)$$

é conosciuto come "*Test degli estremi*". 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) é nota su variabili random uniformi in (0,1) indipendenti e identicamente distribuiti.
- Valutare la verosimiglianza del valore computato del test statistico con la relativa distribuzione teorica da cui é stato assunto adottando una metrica basata sulla distanza lineare.

Questo test si basa sulla seguente considerazione:

Teorema Se U_0, U_1,...,U_{d-1} é una sequenza di variabili *Uniform(0,1)* e se

$$\mathrm{R} = \max U_0, U_1,...,U_{d-1}$$

allora la variabile U = R^d é una *Uniform(0,1)* [1].

In pratica tale test verifica che la variabilità delle altezze dell'istogramma prodotto dai numeri pseudo-casuali generati é sufficientemente piccola da poter concludere che questi numeri appartengano ad una popolazione *Uniform(0,1)*. Questa operazione é fondamentale in quanto tutte le altre distribuzioni di probabilità contenute all'interno della libreria utilizzata vengono generate a partire dalla *Uniform(0,1)*.

[1] Attenzione: il teorema afferma che la variabile U é una Uniform(0,1), mentre la variabile R non lo é.

1.3 Algoritmo

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, elevando tale massimo all d-esima potenza e conteggiando tutti i massimi generati in un array. Tale funzione viene applicata per ogni stream del generatore Lehmer. Tutto ciò é illustrato di seguito:

```
for(stream = 0; stream < 256L; stream++) {
    long o[K];
    memset(o, 0, K * sizeof(long));
    for(i = 0; i < N; i++) {
        double r = Random();
        for(j = 1; j < D; j++) {
            u = Random();
            if(u > r) r = u;
        }
        u = exp(D * log(r));
        x = u * K;
        o[x] ++;
    }
}
```

Per ogni stream, si determina quindi la variabile chi-quadro v tramite questa funzione:

```
    for(i = 0; i < K; i++) {
    v += (square(o[i] - e_x));
    }
    v /= e_x;
```

I valori critici v_1^* e v_2^* vengono calcolate utilizzando 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 é stato scelto un livello di confidenza con parametro $\alpha = 0.05$, mentre i parametri N e K sono rispettivamente N = 100000 e K = N/20 = 5000. 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 probabilitá $1 - \alpha$.

1.4 Test e conclusioni

Il grafico risultante di questo test empirico é illustrato di seguito:

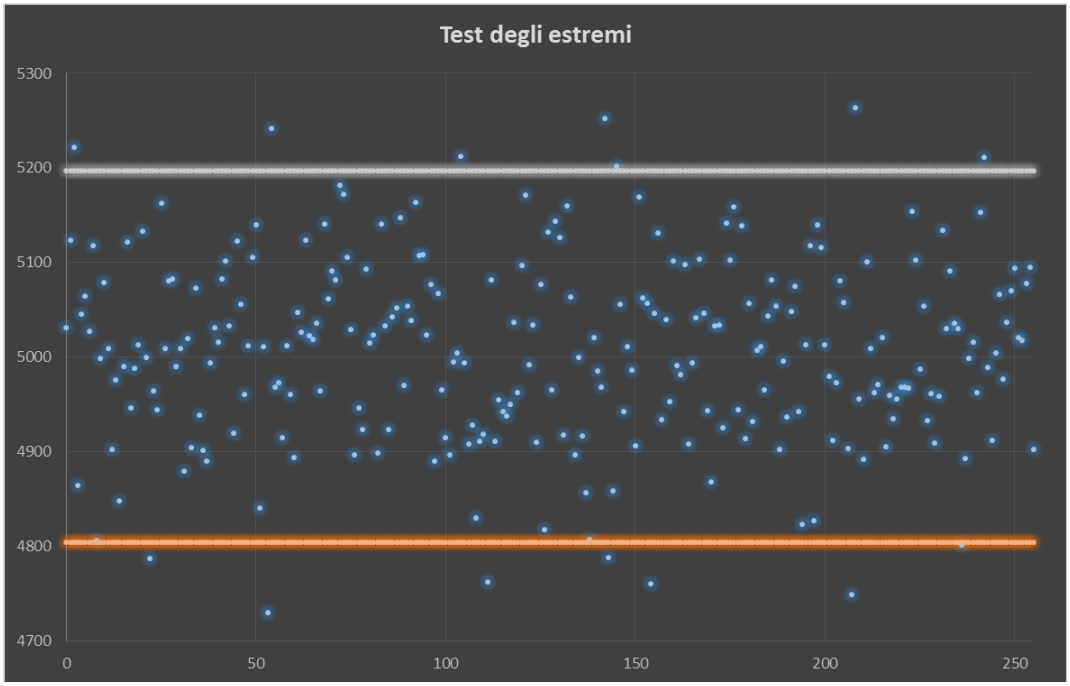


Figura 1.1: Test degli estremi

I valori critici sono visualizzati come linee orizzontali: quella inferiore rappresenta $v_1^* = 4804.92$ mentre quella superiore $v_2^* = 5196.86$

Dalla simulazione effettuata si é notato che il numero di test statistici $v > v_2^*$ sono stati esattamente 7, come il numero di test $v < v_1^*$. Di seguito é riportato l'output del programma:

```
Number of failed: 14
Number of passed: 242
Total number of tests: 256
```

Figura 1.2: Risultati

Considerando il numero totale di test falliti (*upper* e *lower bound*) pari a 14, 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 circa 256 * 0.05 = 13 fallimenti. Questo valore può essere una indicazione della bontà del generatore di Lehmer implementato.

Capitolo 2

Obiettivi

L'obiettivo del progetto quello di modellare, pianificare e sviluppare un simulatore di traffico web che rispetti le specifiche di consegna.

Il sistema reale presenta le seguenti caratteristiche:

- un flusso di utenti che si connettono al sistema sotto forma di sessioni;
- un numero di richieste di cui si compone ogni sessione;
- un front-end server;
- un back-end server con un database.

Il simulatore verrá utilizzato per analizzare il comportamento stazionario relativo ad alcuni indici prestazionali quali il tempo di risposta del sistema, il throughput e la percentuale di sessioni abortite e rifiutate. Il sistema dispone infatti anche di un meccanismo di *gestione del sovraccarico* basato sul monitoraggio in tempo reale dell'utilizzazione del front-end.

Capitolo 3

Modello Concettuale

Il modello sviluppato composto da un ramo principale (comprensivo di Front Server, Back-End Server e relative code) e da una componente di retroazione che si compone di un centro di Client, all'interno del quale gli utenti passano un certo tempo a pensare prima di effettuare la richiesta successiva.

Al sopraggiungere di una nuova sessione, questa verrá processata immediatamente dal Front Server, nel caso in cui la sua coda sia vuota, altrimenti verrá posta in attesa del proprio turno con un conseguente ritardo. Una volta che il Front Server ha elaborato tale sessione, quest'ultima verrá processata all'interno di un Back-End Server, se la coda di quest'ultimo é vuota, altrimenti potrebbe subire un certo ritardo dovuto dall'attesa del proprio turno. Al termine di tale servizio la sessione uscirá dal sistema, nel caso in cui abbia completato tutte le richieste che la componevano, altrimenti verrá reindirizzata nuovamente all'ingresso del sistema attraverso una ramo di feedback.

In tale ramo la sessione permane in un centro di Client in cui l'utente puó spendere del tempo per pensare alla sua richiesta successiva. Dopo tale attesa la sessione tenta di rientrare nel sistema per ricevere un ulteriore servizio, andandosi a posizionare alla fine della coda FIFO del Front Server, inoltre tale servizio é senza prelazione ed é conservativo.

Il sistema verrá regolamentato attraverso un meccanismo di *overload management* che permetterá di limitare i tempi di esecuzione della simulazione, nel caso della distribuzione peggiore. Questo consiste nel monitorare l'utilizzazione del sistema implementato, ovvero una volta che abbia raggiunto l' 85%, il sistema rigetterá tutte le richieste (in arrivo e attive), fino a che l'utilizzazione non raggiunga una soglia inferiore al 75%. Tale tipologia di simulazione garantisce una notevole semplicitá di gestione dell'intero sistema attraverso la facilitá di avanzamento del tempo ed il controllo delle diverse

tipologie di eventi che occorrono durante le varie esecuzioni.

3.1 Variabili di Stato

Il sistema descritto completamente dalle seguenti variabili di stato:

- $busy_{fs}$= stato di occupazione del Front Server
- $queue\_length_{fs}$= numero di richieste nella coda del Front Server
- $busy\_bes$= stato di occupazione nella coda del Back-End Server
- $queue\_length_{bes}$= numero di richieste nella coda del Back-End Server
- $client\_counter$= numero di client attivi in un dato istante di tempo

Al fine di calcolare medie, varianze, intervalli di confidenza e per visualizzare l'avanzare della simulazione sono state utilizzate anche altre variabili (per lo pi booleane o di tipo contatore): *arrivals sessions requests dropped aborted* .

3.2 Modello delle specifiche

Nello sviluppo del modello delle specifiche, l'attenzione 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:

- $\lambda_{sessioni} = 35\frac{richieste}{s}$ (distribuito esponenzialmente)
- $Dimensione_{sessioni} \sim Equilikely(5, 35)$
- $E[Z] = 7s$ (distribuito esponenzialmente)
- $E[D]_{front-end} = 0.00456s$ (distribuito esponenzialmente)
- $E[D]_{back-end} = 0.00117s$ (distribuito esponenzialmente)

3.3 Eventi

Dopo aver prodotto un modello delle specifiche sono stati identificati gli eventi generati dal sistema, durante la simulazione:

- NewSession: una nuova sessione entra nel sistema. Ovvero verrá inserita nel Front Server o nella sua coda qualora quest'ultimo fosse giá occupato;
- FS_Completion: il Front Server evade una richiesta e la invia al Back-End Server;
- BES_Completion: il Back-End Server evade una richiesta. La sessione corrispondente puó dunque essere completata del tutto e quindi uscire dal sistema oppure migrare verso il centro Client nel caso in cui il suo numero di richieste sia non nullo.
- Client_Completion: dopo aver passato un certo tempo in fase di *Thinking*, una sessione lascia il centro Client e rientra nel sistema attraverso il ramo di retroazione, quindi é direttamente inserita nella coda del Front Server.

Capitolo 4

Modello Progettuale

4.1 Architettura

La struttura associata all'Architettura da implementare puó essere rappresentata semplicemente come in figura 4.1

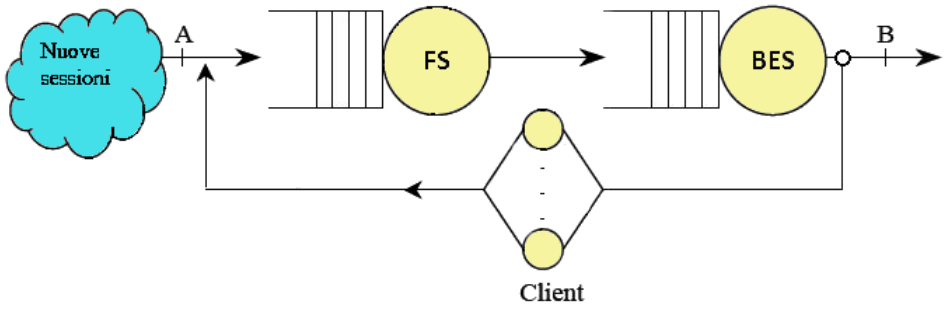


Figura 4.1: Rappresentazione grafica dell'architettura del sistema.

All'inizio della simulazione il primo evento che si verifica é sempre l'arrivo di una nuova sessione (evento NewSession).

I client del centro delle sessioni attive, all'interno del ramo di retroazione sono paralleli ed infiniti, definendo un infinity server.

Ovviamente i nuovi eventi in arrivo verso il sistema verranno posti in coda al FrontServer, nel caso in cui non possano essere serviti. Come é possibile notare nella figura precedente, la sessione una volta servita dal Front Server

verrá posta in coda verso il Back-End Server finché non verrá servita da quest'ultimo. Infine le sessioni saranno completate oppure poste nella zona di "*Thinking*" finché non verranno reinserite nella coda del Front-End in attesa di essere serviti, passando attraverso un ramo di feedback.

4.2 Clock di simulazione e schedulazione di Eventi

Nella fase implementativa si tiene conto dell'avanzamento del tempo per mezzo della variabile *current_time*. Il meccanismo di avanzamento del tempo scelto é il *Next-Event Time Advance*. Questa scelta garantisce che gli eventi occorrano nella sequenza corretta,ovvero vengono processati in ordine crescente rispetto al tempo di schedulazione. Si utilizza, inoltre, il flag di *arrivals* per regolare l'accettazione delle nuove sessioni: se impostata a zero vengono inibiti i nuovi arrivi[1], altrimenti si procede normalmente con la simulazione.

4.3 Event List

Per la gestione degli eventi si utilizza una lista collegata di strutture *Event*, come quella mostrata in figura, salvate in ordine crescente rispetto al tempo. Ogni nodo contiene il tempo di occorrenza e la sua tipologia. Un gestore di eventi é utilizzato per il demultiplexing di tale lista facendo uso di una funzione pop() per ottenere il *Next-Event* da processare.

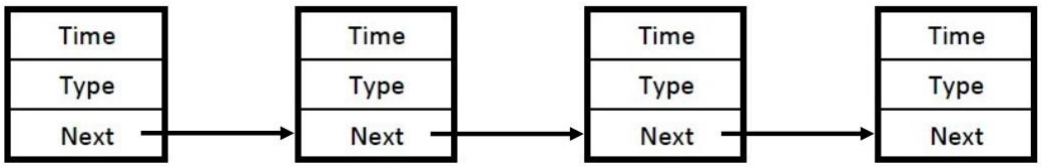


Figura 4.2: Struttura lista eventi

[1]Ovvero viene eseguito il drop della sessione in entrata o l'abort se la sessione é giá nel sistema

4.4 Arrival Queue

Al fine di ottenere informazioni riguardo i tempi di attesa che le sessioni sperimentano durante la loro permanenza nel sistema, si utilizzano delle strutture dati atte a registrare tali informazioni ed utilizzate negli algoritmi del calcolo delle medie. Tali strutture, denominate *ArrivalQueue*, immagazzinano i tempi di arrivo delle sessioni nelle sottosezioni del sistema (ovvero quando una sessione entra nel *Front Server* o nella sua coda, quando entra nel *Back-End Server*, e cosí via). Ad ogni completamento sperimentato da una sessione viene utilizzata la coda relativa e si calcola, per differenza, il tempo effettivo che la sessione ha passato in quella parte di sistema. Tutto ci é possibile grazie all'ipotesi che l'ordine di arrivo, all'interno delle code del sistema, é sempre `preservato`. Infatti la prima sessione ad entrare nella coda del Front Server, ad esempio, sará la prima a lasciarlo.

4.5 Request Queue

Dal momento che é impossibile identificare una singola richiesta utilizzando la Next-Event Simulation, il problema di preservare l'informazione riguardante il numero di richieste attive di cui si compone una sessione viene risolto con la struttura dati *Request Queue*.

Ad ogni richiesta completata si decrementa il contatore delle richieste relative a quella sessione in modo da propagare tale informazione a tutte le richieste future. Quando il contatore arriva a zero la sessione viene completata del tutto e di conseguenza abbandona il sistema.

4.6 Client Order List

Per preservare una Next-Event Simulation priva di contaminazioni derivanti da una possibile aggiunta di dati identificati delle sessione o delle richieste all'interno degli eventi di base, la Client Order List permette di conservare le informazioni relative all'ordine di arrivo e di uscita degli utenti all'interno del centro Client. Attraverso questa informazione é possibile gestire il corretto ordine degli elementi della Request Queue nonostante siano condizionati da una mancanza di determinismo circa l'ordine di completamento degli utenti durante il loro periodo di Think Time.

4.7 Personalizzazione del modello

In base alle scelte effettuate dall'utente nella fase di *setup* possibile decidere di avviare la simulazione:

- senza **Overload Management**
- con **Overload Management**

possibile scegliere quale distribuzione utilizzare tra:

- *Esponenziale*
- *10-Erlang*
- *Iperesponenziale*

É poi possibile impostare i parametri riguardanti lo *STOP iniziale, STOP finale e Numero di Run*. Questi parametri sono utilizzati al fine di calcolare il passo[2] di ogni esecuzione mediante la seguente formula:

$$Step = \frac{StopIniziale - StopFinale}{NumerodiRun}$$

Giunta una nuova sessione, le richieste vengono accodate nel *Front Server* in attesa di essere processate; effettuato il servizio, ovvero verificatosi un evento *FS_COMPLETION*, tale richiesta passa successivamente al servizio del *Back-End Server*, al termine del quale si verifica un'occorrenza del tipo *BES_COMPLETION*.

A questo punto il simulatore valuta per la suddetta sessione il numero di richieste rimanenti ed effettua una scelta: se il numero di richieste per quella sessione é zero allora l'utente esce dal sistema; se, al contrario, cosí non fosse quest'ultimo attraverserá il ramo di feedback per esaurire le proprie richieste rimanenti.

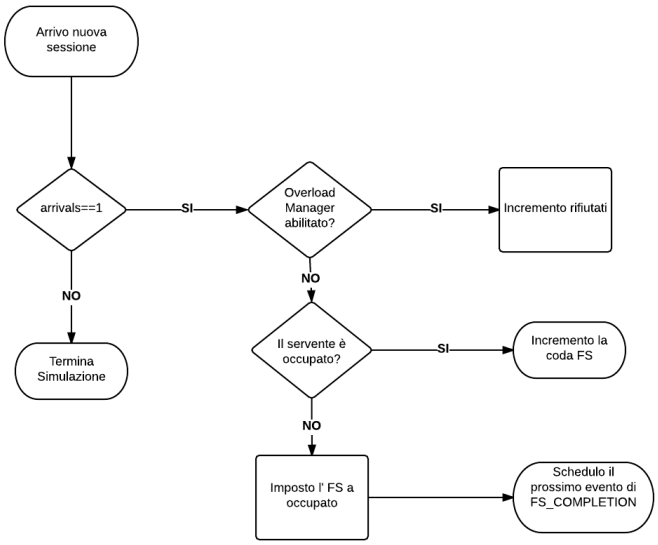
4.8 Avvio e fine simulazione

Come unico vincolo per il termine della simulazione sono utilizzate due variabili, *START* e *STOP*, che regolano gli orari di apertura e chiusura del sistema.
Quest'ultimo inizia a schedulare eventi di tipo *NewSession* solo dopo il tempo di START e inibisce tali eventi all'occorrenza del tempo di STOP FINALE.

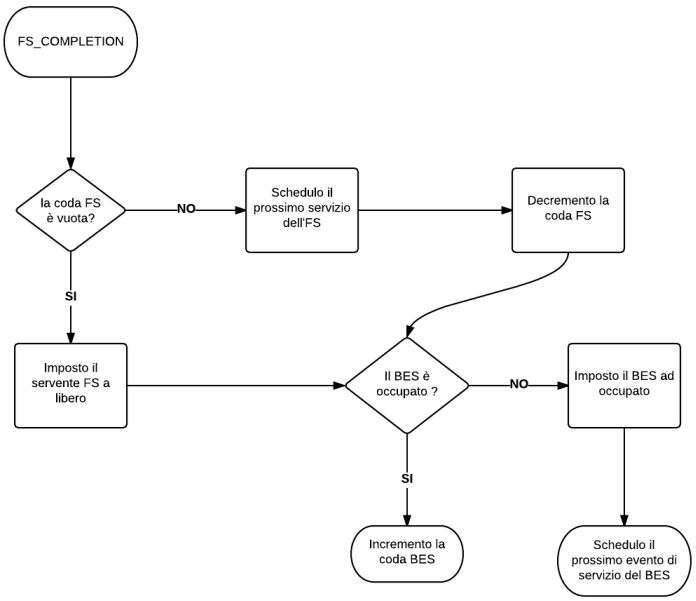
[2]Per "*passo*" si intende l'incremento unitario dell'istante conclusivo di simulazione durante un insieme di run, nel calcolo del comportamento *steady-state*

4.9 Algoritmi di Gestioni eventi

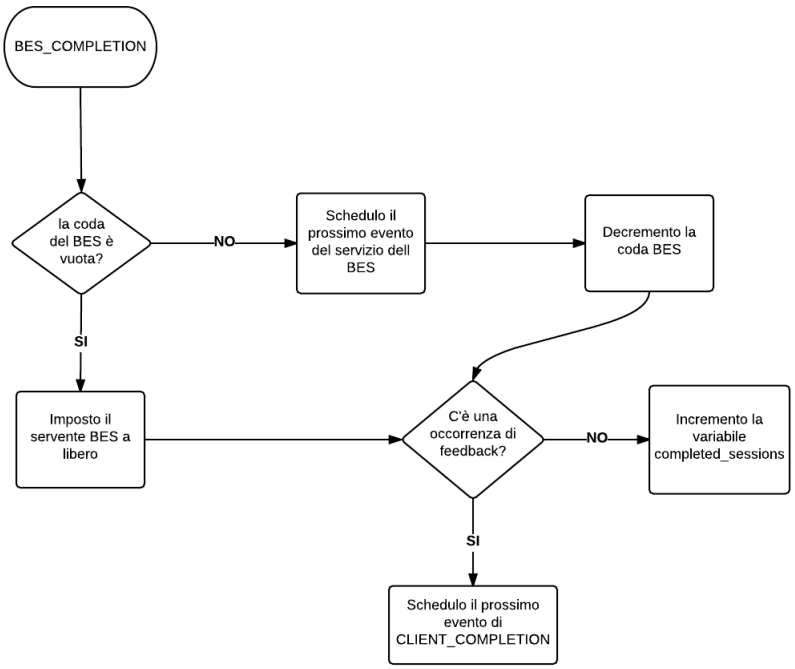
4.9.1 NewSession



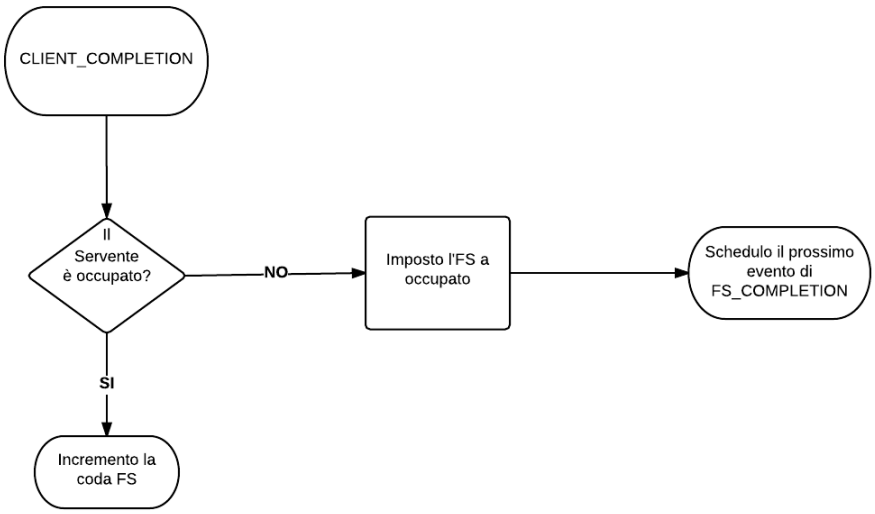
4.9.2 FS Completion



4.9.3 BES Completion



4.9.4 Client Completion



4.10 Dati esaminati

Le richieste progettuali hanno sancito la raccolta di alcuni dati definite da alcune *metriche* quali:

- **Troughput**: é l'indice che misura il totale di sessioni completate nell'unitá di tempo. Tale indice viene calcolato attraverso il rapporto tra le sessioni totali processate dal sistema e l'intervallo di tempo necessario per ultimare questo compito. Il troughput basato sulle sessioni é un ottimo indice per valutare il numero di utenti serviti dal sistema in un intervallo dato, risultando una misura sensibile per l'utente finale. Per tener conto della capacitá computazionale del sistema data dal numero di singole richieste completate, si é deciso di calcolare anche il troughput basato sulle richieste attraverso un semplice rapporto tra il totale delle richieste processate dal sistema e l'intervallo di tempo utilizzato nell'eseguirle.

- **Drop Ratio**: misura il rapporto tra il totale delle sessioni rifiutate dal sistema ed il numero di sessioni totali che tentano di entrare nel sistema (accettate + rifiutate).

- **Abort Ratio**: é il rapporto tra le richieste abortite ed il totale di richieste processate dal sistema. Questo é dovuto dal fatto che una richiesta puó rientrare nel sistema, tuttavia in condizioni di saturazione, tale richiesta non é in grado di poter rientrare all'interno del Front Server, quindi viene appunto abortita.

- **Tempo di Risposta del Sistema**: viene inteso come la somma del tempo di risposta (Tempo in coda + Tempo di Servizio) del front-end e del back-end. In pratica il tempo di risposta é inteso come il tempo che intercorre tra l'uscita di una richiesta dal centro di Client, ovvero la fine del periodo di "*Thinking*" di un dato utente, ed il ritorno di tale richiesta al centro.

4.11 Modello Analitico

Al fine di prevedere, in linea di massima, i risultati del simulatore, viene elaborato un modello analitico semplificato per poter studiare il sistema preso in esame. Il sistema implementato presenta numerosi vincoli ed una complessitá insita nelle specifiche, a tal proposito viene proposto un modello volontariamente e lievemente differente dal caso reale, fornendo un limite inferiore delle prestazioni e degli indici misurati.

Il sistema esaminato risulta essere interattivo, in quanto avviene uno scambio di richieste tra Client e Server, con un comportamento a rete aperta (le nuove sessioni possono entrare nel sistema qualora questo non fosse saturo), a tal proposito si é deciso di presentare due modelli differenti tra loro, uno a rete aperta ed uno a rete chiusa. Questi permettono di descrivere in dettaglio la maggior parte degli elementi che costituiscono il sistema stesso.

4.12 Modello semplificato a rete aperta

In questo primo modello si ha una rete aperta di Jackson con un tasso di sessioni in arrivo pari a γ . Si é deciso cosí, per questioni di semplificazione del modello, di non inserire, come componenti, la Priority Queue e il centro di Client, in quanto risultavano di difficile analisi in un modello di rete aperta come quello di Jackson.

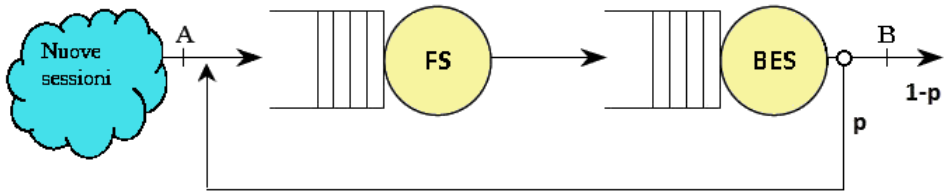


Figura 4.3: Modello a rete aperta

$\gamma = 35 sessioni/s$

$$\begin{cases} \lambda_1 = \gamma + p\lambda_2 \\ \lambda_2 = \lambda_1 \end{cases} \rightarrow \begin{cases} \lambda_1(1-p) = \gamma \\ \lambda_2 = \lambda_1 \end{cases} \rightarrow \begin{cases} \lambda_1 = \frac{\gamma}{(1-p)} \\ \lambda_2 = \frac{\lambda_2}{(1-p)} \end{cases} \quad \lambda_1 = \lambda_2 = \frac{\gamma}{(1-p)}; p = \frac{19}{20} \rightarrow \lambda_1 = \lambda_2 = 35 \times 20 = 700 richieste/s$$

$\lambda_1 \gg \mu_{FS}; \lambda_2 = \mu_{FS} poich \mu_{BES} \gg \lambda_1$ Viene quindi calcolato il throughput, ovvero il numero di sessioni che escono dal sistema al secondo:

$$\lambda_2 = \frac{\lambda_2}{(1-p)} = \frac{\mu_{FS}}{20} \approx \frac{219}{20} richieste/s = X_{sessioni} \quad X_{richieste} = X_{sessioni} \times 20$$

Per quanto concerne l'indice riguardante la percentuale di sessioni rifiutate dal sistema, si trova:

$dropped = \frac{\#sessioniaccettate}{\#totalearrivi} \rightarrow \frac{35 - X_{sessioni}}{35} = 1 - \frac{2}{7} \approx 0.7 \rightarrow \%dropped \approx 70\%$

4.13 Modello semplificato chiuso

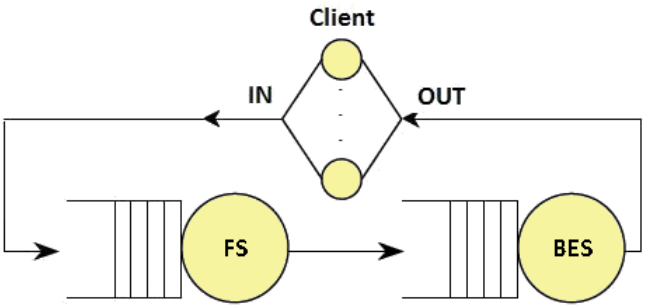


Figura 4.4: Modello a rete chiusa

$D_{FS} = 0.00456; D_{BES} = 0.00117; E[Z] = 7s; N = 250$ (in realtá il numero aumenta costantemente!). Anche qui si puó calcolare il throughput del sistema, come: $X = min\{\frac{1}{D_{max}}, \frac{N}{D+Z}\} = min\{\frac{1}{D_{FS}}, \frac{250}{D_{FS}+D_{BES}+Z}\} = 35.685 richieste/s$. Da cui si puó ricavare il lower bound per il tempo medio di risposta. $E[R] \geq max\{D, \frac{N}{X} - E[Z]\} = max\{0.00573, \frac{250}{35} - 7\} \approx 0.0057447$

Capitolo 5

Modello Computazionale

Il programma realizzato é composto da un file eseguibile (simulatore.c) e di alcuni file dove sono contenute funzioni di appoggio. (*event_list*, *arrival_queue*, *autocorrelation*, *client_req*, *req_queue*). Il software sviluppato é codificato con il linguaggio *C*.

5.1 Simulatore.c

Questo applicativo é il cuore del simulatore implementato. Tale programma sfrutta un'interfaccia testuale per interrogare l'utente circa la configurazione da adottare per la simulazione da eseguire. É possibile selezionare diverse opzioni:

- Scegliere la distribuzione da testare tra:
  - *Esponenziale*
  - *HyperEsponenziale*
  - *10-Erlang*
- Scegliere il seed per la generazione di numeri random
  - 615425336
  - 37524306
  - 123456789
- Scegliere i parametri della simulazione:
  - *Primo Stop*

  - *Ultimo Stop*
  - *Numero di esecuzioni*

- La possibilitá di stampare sulla console le variabili pi importanti durante la simulazione.

Alla luce di quanto illustrato in precedenza si puó sottolineare la capacitá del programma di poter calcolare automaticamente il passo della simulazione: l'utente dovrá soltanto inserire il parametro iniziale e finale della simulazione ed il numero di run che desidera eseguire, il software genererá il passo unitario per il ciclo di simulazione. Inoltre all'interno del programma é stato implementato un meccanismo di *Overload Management*, che l'utente puó abilitare o disabilitare, applicato poi nella distribuzione della *10-Erlang*. I risultati prodotti da questa simulazione vengono trascritti su un file di tipo "*.csv*" in modo da dare all'utente una chiara ed equilibrata visione dei dati ottenuti.

5.2 Event List

La lista di eventi é costituita da strutture di tipo *Event*, formate da un campo di tipo *double*, che indica il time che rappresenta il tempo di occorrenza, un *_EVENT_TYPE* type rappresentante il tipo di evento ed un puntatore *next* alla struttura seguente. Per merito della funzione *add_event()* é possibile aggiungere eventi alla lista. Verranno inseriti seguendo un ordinamento crescente rispetto alla variabile time. Durante l'inserimento dei dati viene effettuato un controllo sulla consistenza dei dati, cio: si controlla, con una funzione *event_check()*, che il tempo sia un valore positivo, che il tipo di evento sia compreso tra 0 e 3. Gli eventi vengono estratti dalla struttura attraverso la funzione *pop_event()*, che preleva l'evento in testa alla lista, restituendolo alla funzione chiamante.

5.3 Arrival queue

La coda di arrivi contiene la struttura di dati di riferimento, che permette la gestione degli arrivi nei vari centri quali: *Front Server*, *Back-End Server*, *Centro Client*. La coda viene rappresentata con una semplice struttura dati formata da: un tempo di arrivo e un puntatore all'elemento successivo. Le funzioni messe a disposizione per questa struttura di dati sono: *arrival_add()*, *arrival_pop()* e l'*arrival_print()*.

- *arrival_add()*: genera un nuovo elemento contenente il tempo di arrivo in un determinato centro,scorre tutto il contenuto della lista generata, posizionando il nuovo nodo in fondo alla coda.

- *arrival_pop()*: permette l'estrazione dell'elemento meno recente della coda.

- *arrival_print()*: stampa lo stato della coda.

5.4 Request queue

La coda delle richieste si basa sul concetto di richiesta: ogni sessione, una volta ammessa all'interno del sistema, definisce un numero di richieste compreso tra 5 e 35. Questa informazione viene inserita all'interno della coda delle richieste in modo da poter sfruttare i dati generati, per modellare gli utenti attivi durante la simulazione. Sfruttando la funzione *enqueue_req()* é possibile inserire tutte le richieste generate dalla nuova sessione vigente. Per poter rimuovere tale dato é possibile utilizzare la *dequeue_req()*. Per verificare l'andamento di tale struttura dati é stata implementata la funzione *print_req()*.

5.5 Client request

La client_req.h implementa una struttura dati in grado di propagare l'informazione circa il numero di richieste relative ad una sessione. Questa viene impiegata all'ingresso e all'uscita dal centro di client in quanto l'ordine di entrata all'interno della zona del *Think Time* é differente da quello di uscita. Per quanto riguarda le funzioni, implementate all'interno di questo programma hanno le stesse funzionalitá espresse nelle sezioni precedenti, sono: *add_client_req()*, *pop_ClientReq()*, *print_client_req()*.

5.6 File Manager

La parte relativa al file manager gestisce tutto il flusso di dati da trascrivere su un file di formato "*.csv*". Composto da tre funzioni:

- *get_date()*: permette di ottenere l'orario e la data correnti, da salvare sul file desiderato.

- *open_file()*: funzione utilizzata per la creazione e l'apertura del file da salvare. Il nome del file viene elaborato utilizzato la funzione **get_date()**, quindi con la data corrente della creazione pi il tipo di distribuzione scelta durante la fase di setting.

- *close_file()*: funzione adottata per chiudere il file una volta terminata la scrittura su di esso.

5.7 Utils

Il file utils.h contiene delle funzioni utilizzate per la pulizia della console e alcune funzioni di scrittura su file di tipo *.csv*, circa i dati ottenuti al termine di una simulazione completata.

5.8 Salvataggio dei dati

Il programma "**simulatore**" serializza tutta le informazioni, generate durante la simulazione, utili al calcolo delle grandezze medie su un file, inserendo blocchi di dati al termine di ogni run. Il nome dei file é legato alla data ed all'orario di avvio della simulazione e al tipo di distribuzione adottata. Il tipo di file che viene generato é di tipo *.xls*. Questi sono adibiti alla creazione automatica dei grafici. Il programma quindi generata automaticamente un file con delimitatori di cella e di riga al fine di consentire all'utente una facile consultazione dei dati ottenuti o semplicemente una facile creazione dei diagrammi. Questi file verranno adottati durante la fase di analisi dei risultati.

5.9 Autocorrelazione

Al fine di calcolare l'autocorrelazione sui tempi di risposta del sistema con **LAG** pari a 20 é stato scritto un programma C che prende in input il file generato dal simulatore e restituisce i valori calcolati delle autocorrelazioni. Per il calcolo di queste si é usata la formula:

$$r_j = \frac{c_j}{c_0} conj = 1, 2, ...20$$

5.10 Intervalli di confidenza

É stato sviluppato un programma scritto in linguaggio **C** allo scopo di calcolare gli intervalli di confidenza di livello 1 - α, dove α pari al 5%. Tale programma prende in input un file in cui sono scritti tutti i valori su cui si vuole fare il calcolo e computa le seguenti statistiche:

- Calcolo della media: $\bar{x}_i = \frac{1}{i}(x_i - \bar{x}_{i-1})$;
- Calcolo della varianza: $v_i = v_{i-1} + (\frac{i-1}{i})(x_i - \bar{x}_{i-1})^2$;
- Calcolo del valore critico: $t^* = idfStudent(n-1, 1-\frac{a}{2})$;
- Calcolo degli estremi dell'intervallo: $\bar{x} \pm \frac{t^* s}{\sqrt{n-1}}$;

Capitolo 6

Verifica

La fase di verifica é molto importante poiché consente di dimostrare la consistenza del programma creato con il modello delle specifiche.

In primis, é stata utilizzata una funzione di stampa per verificare il corretto flusso delle sessioni all'interno dell'intero sistema. Tale funzione, la **print_system_state()**, stampa le statistiche piú importanti del sistema in tempo reale su standard output.

In secundis, si é notato e verificato che le liste contenenti le sessioni e le richieste si riempiono e si svuotano in modo corretto. Anche in questi casi sono stati necessarie funzioni di stampa per verificare che l'aggiornamento fosse adeguato.

In terzis, i vincoli sullo stato del sistema e sull'entrata in azione dell'overload manager, e sul calcolo delle medie sono tutti soddisfatti.

Il simulatore parte con il numero di sessioni nullo e tutte le variabili di stato e di supporto sono inizializzate opportunamente. Il meccanismo di step consente di avere una stampa aggiornata di tutti i parametri rilevanti, come throughput e tempo di risposta del sistema ad esempio, per capire in quale momento ad esempio si entra in uno stato stazionario.

Il numero di sessioni rifiutate e abortite cresce in maniera consistente con il meccanismo di controllo delle ammissioni.

Infine, come ultima verifica, é stato dimostrato che superato il tempo di *STOP finale*, chiamato *FIN* all'interno del codice, nessuna nuova sessione fosse accettata dal sistema e, successivamente a tale istante la simulazione viene interrotta poich la coda tende a crescere all'infinito.

Capitolo 7

Validazione

In questa fase viene testata la corrispondenza del simulatore con il modello reale. Sono stati effettuati i seguenti controlli:

- Saturazione del Front Server: ci sono due concause che portano alla saturazione del Front Server, in primo luogo c'é una enorme differenza tra il tasso di servizio di quest'ultimo e quello del Back-End Server che é circa 3 volte maggiore; inoltre il numero di utenti che stazionano nel centro Client tende a crescere enormemente e tale comportamento influenza l'andamento di richieste in entrata al Front Server che ne determina la congestione.

- La coda del Back-End Server risulta essere quasi sempre vuota poiché é il Front Server che elabora in maniera lenta mentre il tasso di servizio in questo caso é di oltre 800 richieste / secondo; questo non permette la creazione di coda e di conseguenza l'utilizzazione é molto bassa.

- Il numero di client é notevolmente elevato: questo é dovuto all'elevato Think Time sperimentato dagli utenti, circa 7 secondi a client. Considerando il throughput del sistema ed i tassi di servizio dei due serventi é immediato notare come il numero dei client attivi contemporaneamente tenda a crescere nel tempo. Tale componente modellato come un Infinite Server.

- Nel caso di overload management la percentuale delle connessioni rifiutate e abortite é molto alta arrivando a toccare anche l'87% nel secondo caso, mentre nel primo é piú contenuta, con un massimo che si attesta circa attorno al 30% per poi oscillare in quell'intorno. Questo perché il Front Server rappresenta il collo di bottiglia dell'intero sistema.

Capitolo 8

Progettazione degli esperimenti

Gli esperimenti eseguiti sono stati i seguenti:

1. Simulazione del sistema quando il Front Server ha distribuzione esponenziale;
2. Simulazione del sistema quando il Front Server ha distribuzione 10-Erlang;
3. Simulazione del sistema quando il Front Server ha distribuzione Iperesponenziale;
4. Simulazione del sistema quando il Front Server ha distribuzione 10-Erlang con soglia di utilizzazione pari all'85%.

Gli esperimenti sono stati basati sui seguenti parametri:

- SEED : indica il seed scelto dall'utente nella fase di setup;
- TIME : durata della simulazione, ovvero lo STOP finale;
- STEP : indica ogni quanto tempo vengono calcolate le statistiche;
- THRESHOLD : indica se l'overload manager é attivo.

SEED	TIME	STEP	THRESHOLD
1	5	7	9
1	5	7	9
1	5	7	9

Per Ogni simulazione si é scelto di salvare su file i parametri necessari all'elaborazione dei risultati ogni 100 secondi, per valutare quando il sistema entra in uno stato stazionario e per vedere l'evolversi della situazione nel tempo. Al superamento dell'istante di STOP la simulazione viene interrotta, senza attendere che la coda risulti vuota poiché crescendo esponenzialmente i tempi per farla svuotare sarebbero talmente alti da rendere la simulazione non piú reale.

Capitolo 9

Simulazioni

Tutte le simulazione sono state effettuate su diversi computer, sia Pc desktop che notebook, sfruttando la portabilitá del software in modo tale da generare i dati sia su sistemi operativi *Windows* che *Linux*. La simulazione viene testata su parametri inseriti dall'utente manualmente, permettendo una facile impostazione del programma, che é in grado di lanciare un'esecuzione della durata di diverse ore. Sono state sfruttate macchine aventi processori multi-core al fine di eseguire piú esecuzioni in parallelo. Considerando i costi della simulazione, a causa della mole di dati elaborati, durante la simulazione, e dalle limitate capacitá computazionali dei personal computer, le esecuzioni presentate nelle sezioni seguenti della relazione sono state frutto di lunghe ore trascorse in attesa degli output desiderati.

Capitolo 10

Analisi dei risultati

I grafici presentati sono il frutto delle simulazioni descritte precedentemente. Queste si dividono in quattro sezioni:

- Analisi del caso in cui il Front-end abbia distribuzione esponenziale;
- Analisi del caso in cui il Front-end ha distribuzione 10 Erlang;
- Analisi del caso in cui il Front-end ha distribuzione iperesponenziale;
- Analisi del caso peggiore tra i tre sopra descritti con l'overload manager attivo.

Per ognuno di questi sono riportati i dati relativi a:

1. Tempo di risposta medio del sistema;
2. Throughput;
3. Autocorrelazione.

10.1 Front server esponenziale

10.1.1 Tempo di risposta

Per prima cosa analizziamo il tempo medio di risposta del sistema. Nella figura é rappresentato un grafico con i tempi medi ottenuti ogni 100 secondi di simulazione. Il sistema risulta essere **instabile**, in quanto i tempi crescono in modo praticamente lineare, questo risulta essere conseguenza del fatto che la cosa del Front server tende a crescere all'infinito.

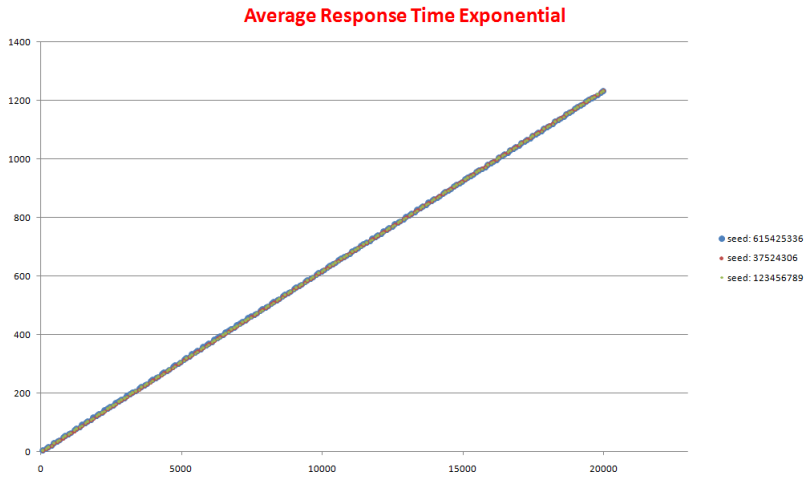


Figura 10.1: Tempo di risposta del sistema nel caso di tempo di servizio esponenziale senza Overload Management.

10.1.2 Throughput

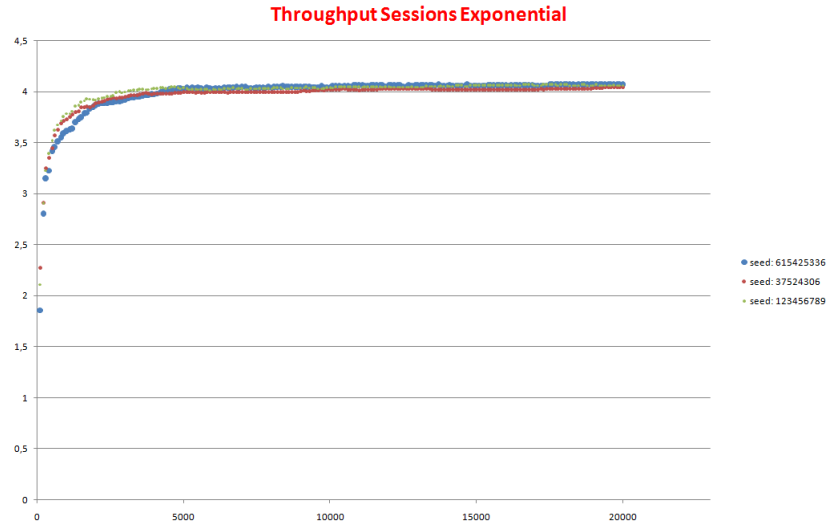


Figura 10.2: Througput del sistema nel caso di tempo di servizio esponenziale senza Overload Management.

La seconda metrica di cui ci andiamo ad interessare é quella relativa al throughput. Come si evince dall'immagine c'é una saturazione molto veloce

del front-end e infatti questo si stabilizza molto velocemente attorno al valore 4.

Un intervallo di confidenza per il throughput calcolato é il seguente:

$$\text{IC} = [3.96532859 - 0.02472767 \ ; \ 3.96532859 - 0.02472767] = [3.94060091 \ ; \ 3.99005626]$$

Mentre qui di seguito vediamo l'istogramma relativo a tale metrica:

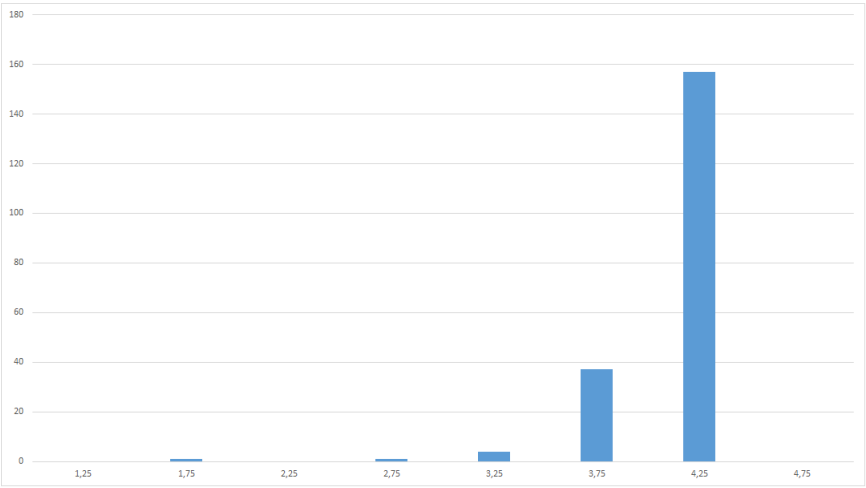


Figura 10.3: Istogramma per il throughput.

10.2 Autocorrelazione

L'ultima metrica da calcolare é quella relativa all'autocorrelazione. Vediamo sotto forma di tabella i risultati ottenuti per i diversi seed.

#	SEED: 615425336	SEED: 37524306	SEED: 123456789
1	0,98995668	0,98997536	0,98994852
2	0,97981444	0,97985091	0,97979832
3	0,96957032	0,96962554	0,96954728
4	0,95922107	0,95930107	0,95919848
5	0,94876988	0,94887862	0,94875186
6	0,93821731	0,9383594	0,93820573
7	0,92757015	0,92773803	0,92755606
8	0,9168206	0,91701647	0,91680759
9	0,90597061	0,90619544	0,90596156
10	0,89501832	0,89527232	0,89501785
11	0,88396744	0,88424369	0,88398003
12	0,87281776	0,81311632	0,87284494
13	0,86157167	0,86189008	0,86161132
14	0,85022491	0,85056456	0,85027646
15	0,83877974	0,83913659	0,83884245
16	0,82723641	0,82760883	0,82730328
17	0,81559788	0,81597844	0,81566353
18	0,80385917	0,80424497	0,80392133
19	0,79202286	0,79241003	0,79208271
20	0,78008154	0,78047206	0,78014424

Tabella 10.1: Tabella esperimenti eseguiti per il front-end esponenziale

10.3 Confronto con altre distribuzioni

Il simulatore come giá detto offre la possibilit di variare la distribuzione del front-end allo scopo di vedere le possibili variazioni di performance del sistema. Le distribuzioni utilizzate sono state le seguenti:

- Esponenziale
- 10 Erlang
- Iperesponenziale

Per confrontare i vari casi di studio é stato utilizzato il tempo di risposta. In questo caso notiamo come ci sia un peggioramento delle condizioni nel caso la distribuzione usata sia la 10 Erlang, mentre nel caso dell'iperesponenziale abbiamo un comportamento molto simile all'esponenziale.

10.4 Overload manager

Osservando i risultati conseguiti nella sezione precedente é stato deciso di attivare il meccanismo di overload management sulla distribuzione 10 Erlang. Questo perché tra le 3 distribuzioni ha mostrato l'andamento peggiore. Si é quindi effettuata una nuova analisi delle metriche prima illustrate.

10.4.1 Tempo di risposta

Il tempo di risposta del sistema quando é attivato questo meccanismo tende a scendere rispetto a prima e risulta arrivare quasi ad una stabilit. Questa stabilit peró non é proprio reale in quanto il sistema *oscilla* attorno a questo valore, anche a causa del fatto che le sessioni giá accettate vengono abortite.

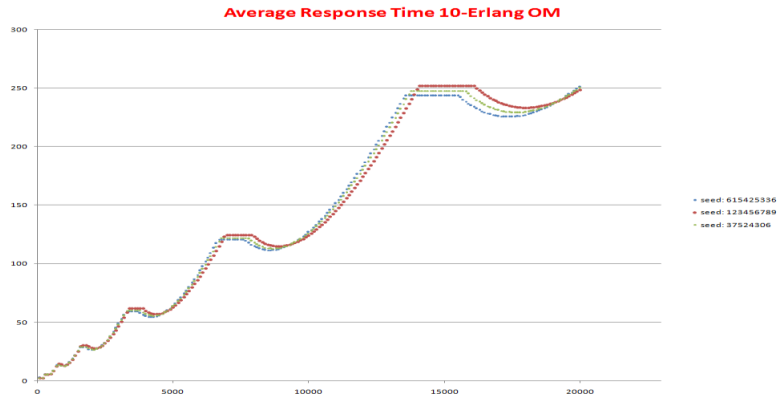


Figura 10.4: Tempo di risposta del sistema nel caso di una 10 Erlang con overload manager.

10.4.2 Throughput

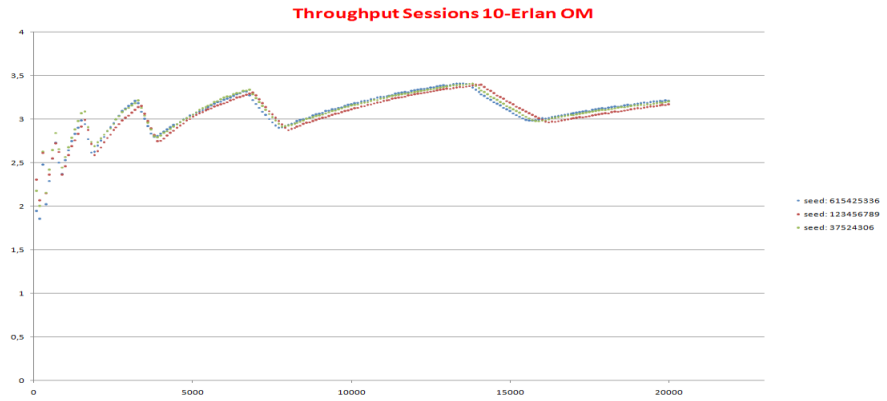


Figura 10.5: Throughput nel caso di una 10 Erlang con overload manager.

Per il throughput osserviamo lo stesso effetto descritto precedentemente in quanto si assesta attorno ad un valore peró senza mai fermarsi del tutto, bensí oscillandovi attorno.

10.4.3 Drop Ratio

Il drop ratio misura il rapporto tra le sessioni rifiutate dal sistema e il numero di sessioni totali (ovvero accettate + rifiutate). Notiamo che questo numero si assesta intorno al 30% e questo accade poiché vengono rifiutate anche le connessioni gi presenti, ció permette al sistema di scendere molto velocemente sotto il 75% di utilizzazione e riprendere dunque un normale funzionamento.

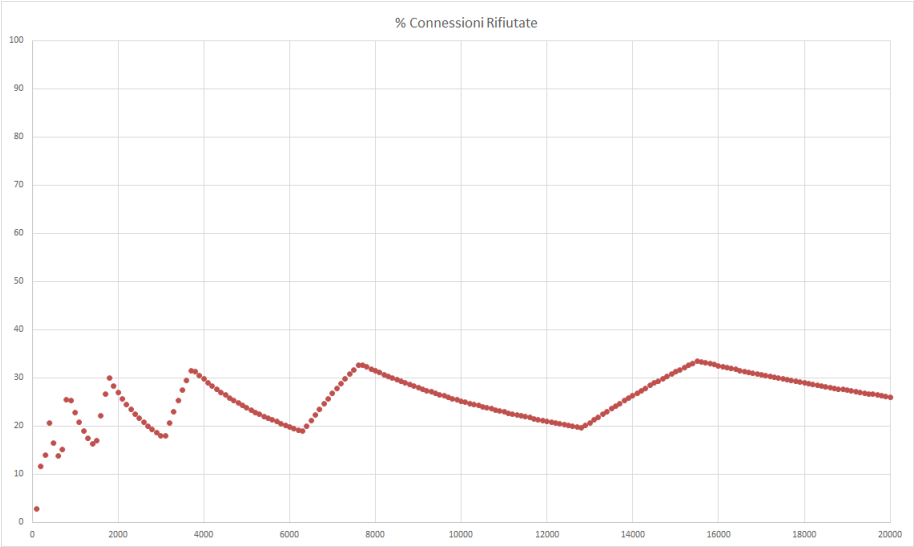


Figura 10.6: Percentuale di connessioni rifiutate dal sistema.

10.4.4 Abort Ratio

Nel caso di overload manager attivo il nostro simulatore deve impedire ad **ogni richiesta** di potersi mettere in coda al Front Server. A causa di questo vincolo l'abort ratio é diverso da zero poiché questo si riferisce alle sessioni che vengono chiuse pur essendo state precedentemente accettate dal sistema. Come si pu vedere dal grafico la percentuale di connessioni abortite dal sistema é abbastanza elevato.

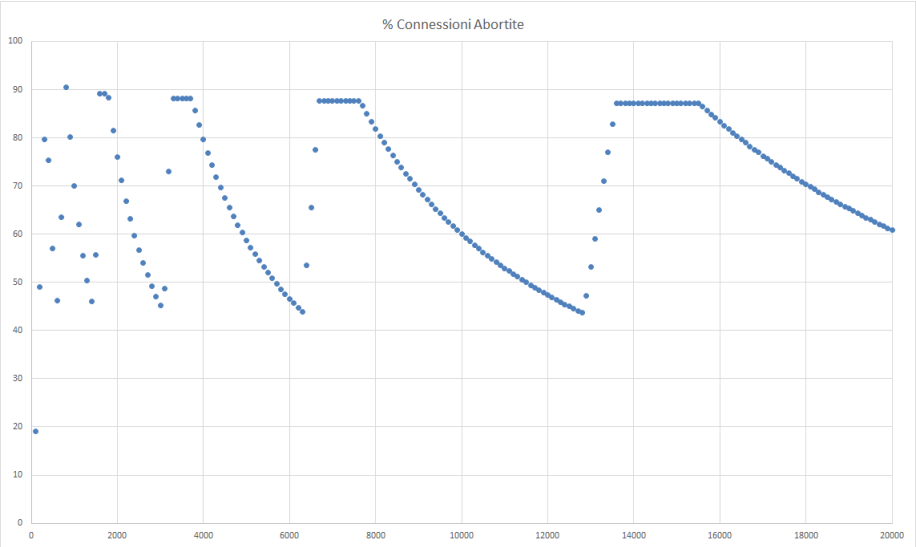


Figura 10.7: Percentuale di connessioni abortite dal sistema.

Appendice A

Codice

A.1 Test degli estremi

A.1.1 test.c

```
/**
   Progetto MPSR
   Test degli estremi
**/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include "rngs.h"
#include "rvms.h"

#define SEED 48271
#define N 100000L
//#define K 5000L
#define K N/20
#define ALPHA 0.05f
#define D 7

#define square(x) (x)*(x)

FILE *openFileForWrite(char *filePath) {
  if(filePath == NULL)
    return NULL;
  return fopen(filePath, "w");
}

void closeFile(FILE *f) {
  if(f != NULL) {
    fclose(f);
  }
}

void writeDataToFile(FILE *f, double v, double v1, double v2, long stream ) {
```

```
  if(f == NULL)
    return;
  if(fprintf(f, "%ld;%.6f;%.6f;%.6f\n", stream, v, v1, v2) < 0) {
    fprintf(stderr, "Failed to write data to file\n");
  }
}

int checkIfTestFailed(double v, double v1, double v2) {
  return v < v1 || v > v2;
}

int main(int argc, char **argv) {
  char *filename = "result.csv";
  long failed = 0L, passed = 0L;
  if(argc > 1) {
    filename = argv[1]; //nome del file
  }

  FILE *dataOut = openFileForWrite(filename);
  if(dataOut == NULL) {
    perror("Data out open");
    exit(EXIT_FAILURE);
  }

  double v1_star = idfChisquare(K - 1, ALPHA/2.0);
  double v2_star = idfChisquare(K - 1, 1.0 - (ALPHA/2.0));

  long stream = 0;
  for(; stream < 256L; stream++) {

    //inizializzazione
    long o[K];

    double v = 0.0;
    double e_x = (double) N / (double) (K);

    //imposta tutti gli intervalli dell'istogramma a 0
    memset(o, 0, K * sizeof(long));

    PlantSeeds(SEED);
    SelectStream(stream);

    //popolo il campione
    long i = 0;
    for(; i < N; i++) {
      double r = Random();
            double u;
            int j;
            for(j = 1; j < D; j++) {
                u = Random();
                if(u > r) r = u;
            }
            u = exp(D * log(r));
      long x = u * K;
      o[x] ++;
    }

    //calcolo la statistica del test
    for(i = 0; i < K; i++) {
      v += (square(o[i] - e_x));
    }
    v /= e_x;
```

```
    writeDataToFile(dataOut, v, v1_star, v2_star, stream);

    //risultati completi
    printf("===Uniformity Test (X~U(0,1))===\n");
    printf("stream ........... = %ld\n", stream);
    printf("seed ............. = %ld\n", SEED);
    printf("alpha ............ = %f\n", ALPHA);
    printf("n ................ = %ld\n", N);
    printf("k bins ........... = %ld\n", K);
    printf("n/k .............. = %f\n", e_x);
    printf("v1* .............. = %f\n", v1_star);
    printf("v2* .............. = %f\n", v2_star);
    printf("v ................ = %f\n", v);
    printf("result ........... = ");

    if(checkIfTestFailed(v, v1_star, v2_star)) {
      printf("FAILED\n");
      failed++;
    } else {
      printf("PASSED\n");
      passed++;
    }
  }

  printf("Number of failed: %ld\n", failed);
  printf("Number of passed: %ld\n", passed);
  printf("Total number of tests: %ld\n", failed+passed);

  closeFile(dataOut);
  return 0;
}
```

A.2 Intervalli di Confidenza

A.2.1 interval_calculator.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Intervalli di confidenza
 */

#define ALPHA 0.05

static double xm_res_time = 0.0;
static double v_res_time = 0.0;
static double n_res_time = 0.0;
static double incr_res_time = 0.0;

static double xm_t_sess = 0.0;
static double v_t_sess = 0.0;
static double n_t_sess = 0.0;
static double incr_t_sess = 0.0;

void set_ic_res_data(double value) {
```

```
    double d = value - xm_res_time;
    n_res_time++;
    v_res_time = v_res_time + d*d*(n_res_time-1)/n_res_time;
    xm_res_time = xm_res_time + d/n_res_time;
}

void set_ic_t_data(double value) {
    double d = value - xm_t_sess;
    n_t_sess++;
    v_t_sess = v_t_sess + d*d*(n_t_sess-1)/n_t_sess;
    xm_t_sess = xm_t_sess + d/n_t_sess;
}

void compute_ic_res() {
    double s = sqrt(v_res_time/n_res_time);
    //Calcolo il critical value
    double t = idfStudent((int) n_res_time-1, 1.0 - (ALPHA/2.0));
    incr_res_time = (t * s) / sqrt((double)(n_res_time-1));
}

void compute_ic_t() {
    double s = sqrt(v_t_sess/n_t_sess);
    //Calcolo il critical value
    double t = idfStudent((int) n_t_sess-1, 1.0 - (ALPHA/2.0));
    incr_t_sess = (t * s) / sqrt((double)(n_t_sess-1));
}

void print_ic_on_file(FILE *g) {
    compute_ic_t();
    compute_ic_res();
    fprintf(g, "\n");
    fprintf(g, "Response time IC = [ %6.8f ; %6.8f ]\n", (xm_res_time - incr_res_time),
         (xm_res_time + incr_res_time));
    fprintf(g, "Throughput IC = [ %6.8f ; %6.8f ]\n", (xm_t_sess - incr_t_sess), (xm_t_sess
         + incr_t_sess));
}
```

A.3 Autocorrelazione

A.3.1 autocorrelation.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Calcolo Autocorrelazione
 */

#define K_CORR 20
#define SIZE_CORR (K_CORR + 1)

static double sum_corr = 0.0;
static double hold_corr[SIZE_CORR];
static double cosum_corr[SIZE_CORR] = {0.0};
static double mean_corr = 0.0;
static long p_corr = 0, j_corr = 0;
```

```
void set_autocorr_data(long point, double average) {
    sum_corr += average;
    if(point < SIZE_CORR) {
        hold_corr[point] = average;
    }
    else {
        for (j_corr = 0; j_corr < SIZE_CORR; j_corr++) {
            cosum_corr[j_corr] += hold_corr[p_corr] * hold_corr[(p_corr + j_corr) %
                SIZE_CORR];
        }
        hold_corr[p_corr] = average;
        p_corr = (p_corr + 1) % SIZE_CORR;
    }
}

void compute_autocorr() {
    int i = 0;
    while (i < SIZE_CORR) {      /* empty the circular array */
        for (j_corr = 0; j_corr < SIZE_CORR; j_corr++) {
            cosum_corr[j_corr] += hold_corr[p_corr] * hold_corr[(p_corr + j_corr) %
                SIZE_CORR];
        }
        hold_corr[p_corr] = 0.0;
        p_corr = (p_corr + 1) % SIZE_CORR;
        i++;
    }
    mean_corr = sum_corr / (batch_num-1);
    for (j_corr = 0; j_corr <= K_CORR; j_corr++) {
        cosum_corr[j_corr] = (cosum_corr[j_corr] / (batch_num - 1 - j_corr)) - (mean_corr *
            mean_corr);
    }
}

void print_autocorr_on_file(FILE *g) {
    fprintf(g, "\n");
    fprintf(g, "Autocorrelation values:\nJ\tVALUE\n");
    for(j_corr = 1; j_corr < SIZE_CORR; j_corr++) {
        fprintf(g, "%3ld\t%6.8f\n", j_corr, cosum_corr[j_corr] / cosum_corr[0]);
    }
}
```

A.4 Simulatore

A.4.1 simulatore.c

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - main function
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <string.h>
```

```c
#include "rng.c"
#include "rvms.c"
#include "global_variables.h"
#include "file_manager.c"
#include "generate_random_value.h"
#include "event_manager.c"
#include "utils.h"
#include "user_signal.c"

void initialize() {
    busy_FS = 0;
    busy_BES = 0;
    FS_counter = 0;
    BES_counter = 0;
    client_counter = 0;
    queue_length_FS = 0;
    queue_length_BES = 0;
    active_client = 0;
    average_res_FS = 0.0;
    average_res_BES = 0.0;
    average_res_client = 0.0;
    FS_utilization = 0.0;
    FS_average_utilization = 0.0;
    completed_sessions = 0;
    opened_sessions = 0;
    requests = 0;
    dropped = 0;
    aborted = 0;
    threshold_exceeded = 0;
    throughput_sessions = 0.0;
    throughput_requests = 0.0;
    arrivals = 1;
    current_time = START;
    prev_time = START;
    ev_list = NULL;
    add_event(&ev_list, GetArrival(START), NEW_SESSION);
    prev_batch_time_completition = START;
    __throughput_sessions = 0.0;
    __throughput_requests = 0.0;
    __FS_utilization = 0.0;
    __FS_average_utilization = 0.0;
    __opened_sessions = 0;
    __completed_sessions = 0;
    __requests = 0;
    __dropped = 0;
    __aborted = 0;
}

void reset() {
    average_res_FS = 0.0;
    average_res_BES = 0.0;
    average_res_client = 0.0;
    FS_utilization = 0.0;
    FS_average_utilization = 0.0;
    completed_sessions = 0;
    opened_sessions = 0;
    requests = 0;
    dropped = 0;
    aborted = 0;
    throughput_sessions = 0.0;
    throughput_requests = 0.0;
```

```
    prev_batch_time_completition = current_time;
}

void compute_statistics() {
    __throughput_sessions += throughput_sessions;
    __throughput_requests += throughput_requests;
    __FS_utilization += FS_utilization;
    __FS_average_utilization += FS_average_utilization;
    __opened_sessions += opened_sessions;
    __completed_sessions += completed_sessions;
    __requests += requests;
    __dropped += dropped;
    __aborted += aborted;
}

void begin_simulation(FILE *graphic) {

    int i = 0;
    initialize();
    for(current_batch = 1L; current_batch <= batch_num; current_batch++) {
        reset();
        // fare qualcosa
        while(completed_sessions < batch_size) {
            // fare tutto
            Event *current = pop_event(&ev_list);
            current_time = current->time;
            FS_utilization += (current_time - prev_time) * (busy_FS);
            FS_average_utilization = FS_utilization /
                 (current_time-prev_batch_time_completition);
            throughput_requests = ((double) requests)/
                 (current_time-prev_batch_time_completition);
            throughput_sessions = ((double)
                 completed_sessions)/(current_time-prev_batch_time_completition);

            manage_event(current);
            prev_time = current_time;

            if(visual_flag == 'Y' || visual_flag == 'y') {
                if(i%300 == 0) {
                    clear_console();
                    // Stampo
                    print_system_state(current->type);
                    i=0;
                    usleep(10000);
                }
                i++;
            }
        }
        print_system_state_on_file(graphic);
        set_ic_t_data(throughput_sessions);
        set_ic_res_data(average_res_FS+average_res_BES);
        printf("\nSimulation completed! (Batch completed: %ld)\n", current_batch);
        compute_statistics();
    }
    // stampa stato finale
    print_final_state(graphic);
    compute_autocorr();
    print_autocorr_on_file(graphic);
    print_ic_on_file(graphic);
}

int main (int argc, char *argv[]) {
```

```
int choice;
long long int SEED = 0;
char t_flag;
FILE *graphic;

if(argc != 1) {
    fprintf(stderr, "Usage: %s\n", argv[0]);
    return EXIT_FAILURE;
}
add_signals();

clear_console();
printf("\nMulti-tier system simulator\n");
printf("-------------------------------------------------------------------------------------------\n");
printf("This system is composed by\n");
printf(" - Front Server\n - Back-end Server\n");
printf("-------------------------------------------------------------------------------------------\n\n");
printf("Choose if Front Server service time has to be:\n");
printf(" 1 - Exponetially distribuited\n");
printf(" 2 - Distributed as a 10-Erlang\n");
printf(" 3 - Distributed as Hyperexponential with p=0.1\n");
printf("Your choice: ");
scanf("%d", &choice);
getchar();
printf("-------------------------------------------------------------------------------------------\n\n");

switch(choice) {
    case 1:
        type = FE_EXP;
        break;
    case 2:
        type = FE_ERL;
        break;
    case 3:
        type = FE_HYP;
        break;
    default:
        printf("Errore. Nessuna distribuzione per il frontend selezionata.\n");
        return EXIT_FAILURE;
}

printf("Choose if threshold should be active:\n");
printf("Your choice: ");
scanf("%s", &t_flag);
getchar();
if(t_flag == 'Y' || t_flag == 'y') {
    printf("Threshold enabled\n");
    threshold_flag = 1;
}
else {
    printf("Threshold disabled\n");
    threshold_flag = 0;
}
printf("-------------------------------------------------------------------------------------------\n\n");

printf("Please select a SEED: \n");
printf(" 1 - 615425336\n");
printf(" 2 - 37524306\n");
printf(" 3 - 123456789\n");
printf(" 4 - Insert another SEED\n");
printf("Your choice: ");
```

```
scanf("%d", &choice);
getchar();

switch(choice) {
    case 1:
        SEED = 615425336;
        break;
    case 2:
        SEED = 37524306;
        break;
    case 3:
        SEED = 123456789;
        break;
    case 4:
        printf("Enter your SEED: ");
        scanf("%lld", &SEED);
        getchar();
        break;
    default:
        printf("Errore. Nessuna distribuzione per il frontend selezionata.\n");
        return EXIT_FAILURE;
}

printf("Chosen SEED is: %lld\n", SEED);
printf("-----------------------------------------------------------------------------------------\n\n");

PutSeed(SEED);

printf("Insert simulation parameters:\n");
printf("Batch size (b): ");
scanf("%ld", &batch_size);
getchar();
printf("Number of batches (k): ");
scanf("%ld", &batch_num);
getchar();

printf("The inserted parameters are: %ld %ld\n", batch_size, batch_num);
printf("-----------------------------------------------------------------------------------------\n\n");

printf("Would you like to see the \"system state\" during the simulation?\n");
printf("Choose [Y/N]:");
scanf("%c", &visual_flag);
getchar();
if(visual_flag == 'Y' || visual_flag == 'y')
    printf("You choose to displat system state\n");
else
    printf("You choose NOT to displat system state\n");
printf("-----------------------------------------------------------------------------------------\n\n");

printf("Press ANY key to start simulation\n");
getchar();

//Aprire i file per la simulazione e iniziare a scriverci dentro
graphic = open_file();
print_initial_settings(graphic, SEED, batch_size, batch_num);

begin_simulation(graphic);

//chiudere i file
close_file(graphic);

return EXIT_SUCCESS;
```

```
}
```

A.4.2 arrival_queue.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Coda di arrivi
 */

struct ArrivalNode{
    double time;
    struct ArrivalNode* next;
};

typedef struct ArrivalNode ArrivalNode;

void arrival_add(ArrivalNode** list, double t){
    if(list == NULL) {
        printf("Errore: la lista  NULL\n");
        return;
    }
    ArrivalNode* temp = (ArrivalNode*) malloc(sizeof(ArrivalNode));
    temp->time = t;
    temp->next = NULL;
    if(*list == NULL){
        *list = temp;
        return;
    }

    ArrivalNode* curr = *list;
    while(curr->next != NULL){
        curr = curr->next;
    }
    curr->next = temp;
}

double arrival_pop(ArrivalNode** list){
    if(list == NULL || *list ==NULL) {
        printf("ArrivalQueue is empty\n");
        return -1.0;
    }
    ArrivalNode* curr = *list;
    *list = (*list)->next;

    double t = curr->time;
    free(curr);

    return t;
}

void arrival_print(ArrivalNode** list){
    if(list == NULL || *list ==NULL) {
        printf("ArrivalQueue is empty\n");
        return;
    }
    ArrivalNode *curr=*list;
    printf("[+] Arrival Queue:\n");
```

```
    while(curr != NULL){
        printf("[Time: %6.4f] -- ",curr->time);
    curr = curr->next;
    }
}
```

A.4.3 client_req.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Coda delle richieste del client
 */

struct ClientReq {
  double time; // time of Client completion
  unsigned int req; // number of remaining requests
  struct ClientReq *next;
};

typedef struct ClientReq ClientReq;

void print_client_req(ClientReq **list) {
  if(*list == NULL) {
    printf("ClientOrderList is empty\n");
    return;
  }
  ClientReq *curr = *list;
  printf("[+] ClientOrderList contains:\n");
  while(curr != NULL) {
    printf("[Time: %6.4f, Rem_Req: %d]\n", curr->time, curr->req);
    curr = curr->next;
  }
}

void add_client_req(ClientReq **list, double x, unsigned int r) {
    if(list == NULL) {
        printf("NULL Pointer to ClientOrderList\n");
        return;
    }

  ClientReq *new_ClientReq = (ClientReq *)malloc(sizeof(ClientReq));
  new_ClientReq->time = x;
  new_ClientReq->req = r;
  new_ClientReq->next = NULL;
  if(*list == NULL) { // empty list case
  //  printf("[T] Empty List\n");
    *list = new_ClientReq;
    return;
  }
  if((*list)->time >= x) { // first element replacement case
  //  printf("[T] First element replacement\n");
    new_ClientReq->next = *list;
    *list = new_ClientReq;
  }
    //  printf("[T] Default add case\n");
    ClientReq *curr = *list;
    while(curr->time < x) {
```

```
    // printf("\n[T] curr->time: %d", curr->time);
        if(curr->next == NULL || curr->next->time >= x) {
            new_ClientReq->next = curr->next;
            curr->next = new_ClientReq;
            return;
        }
        else curr = curr->next;
    }
}

// Return the first element and remove it
ClientReq* pop_ClientReq(ClientReq** head) {
    if(head == NULL || *head == NULL) return NULL;
    ClientReq* temp = *head;
    *head = (*head)->next;
    return temp;
}
```

A.4.4 event_list.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Tipologia di Eventi
 */

#define MIN_TYPE NEW_SESSION
#define MAX_TYPE CL_COMPL

enum _EVENT_TYPE {
    NEW_SESSION,
    FS_COMPL,
    BES_COMPL,
    CL_COMPL,
    SYSTEM_PRINT
};

typedef enum _EVENT_TYPE EVENT_TYPE;

struct Event {
  double time;
  EVENT_TYPE type;
  struct Event *next;
};

typedef struct Event Event;

int event_check(double t, EVENT_TYPE type) {
    return t>=0.0 && (type >= MIN_TYPE && type <= MAX_TYPE);
}

void printlist(Event **list) {
  if(*list == NULL) {
    printf("List is empty\n");
    return;
  }
  Event *curr = *list;
  printf("[+] List contains:\n");
```

```
  while(curr != NULL) {
    printf("[Type: %d, Time: %6.4f)\n", curr->type, curr->time);
    curr = curr->next;
  }
}

void add_event(Event **list, double x, EVENT_TYPE typ) {
    if(list == NULL) {
        printf("NULL Pointer to list\n");
        return;
    }
    if(!event_check(x, typ)) {
        printf("[+] Event check failed. Please review your data.\n");
        return;
    }
  Event *new_Event = (Event *)malloc(sizeof(Event));
  new_Event->time = x;
  new_Event->type = typ;
  new_Event->next = NULL;
  if(*list == NULL) { // empty list case
  //  printf("[T] Empty List\n");
    *list = new_Event;
    return;
  }
  if((*list)->time >= x) { // first element replacement case
  //  printf("[T] First element replacement\n");
    new_Event->next = *list;
    *list = new_Event;
  }
    //  printf("[T] Default add case\n");
    Event *curr = *list;
    while(curr->time < x) {
    //  printf("\n[T] curr->time: %d", curr->time);
        if(curr->next == NULL || curr->next->time >= x) {
            new_Event->next = curr->next;
            curr->next = new_Event;
            return;
        }
        else curr = curr->next;
    }
}

// Return the first element and remove it
Event* pop_event(Event** head) {
    if(head == NULL || *head == NULL) return NULL;
    Event* temp = *head;
    *head = (*head)->next;
    return temp;
}

char* event_translator(EVENT_TYPE t){
  switch(t){
    case NEW_SESSION:
           return "New session";
    break;
    case FS_COMPL:
           return "Front server completition";
    break;
    case BES_COMPL:
           return "Back-end server completition";
    break;
    case CL_COMPL:
```

```
            return "Client completition";
    break;
        case SYSTEM_PRINT:
            return "User forced to print output";
    break;
    default:
            return "Error event";
    break;
  }
  return "Wrong code!\n";
}
```

A.4.5 event_manager.c

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Processamento degli eventi
 */

long FS_counter, BES_counter, client_counter;
int queue_length_FS, queue_length_BES, busy_FS, busy_BES, active_client;
double average_res_FS, average_res_BES, average_res_client;
int threshold_exceeded;

// NewSession Management
void NewSession(Event* ev) {

    // If current time is not beyond STOP, a NewSession is scheduled
    if(arrivals) {
        double new_time = GetArrival(ev->time); // compute new session arrival time
        add_event(&ev_list, new_time, NEW_SESSION); // create NewSession event and schedule
             it
    }

    if(threshold_flag) {
        if(threshold_exceeded) {
            if(FS_average_utilization <= THRESHOLD_MIN)
                threshold_exceeded = 0;
            else {
                dropped++;
                return;
            }
        }
        // la 10-Erlang  la peggiore implemento quindi il Threshold all' 85%
        else if(FS_average_utilization >= THRESHOLD_MAX) {
            threshold_exceeded = 1;
            dropped++;
            return;
        }
    }
    opened_sessions++;
    enqueue_req(&req_queue, GetRequests());
    arrival_add(&arrival_queue_FS, ev->time);
    if(busy_FS) queue_length_FS++; // if server is busy, add 1 to its queue
    else {
        busy_FS = 1;
        switch(type) {
```

```
            case FE_EXP:
                add_event(&ev_list, GetExponentialServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
            case FE_ERL:
                add_event(&ev_list, GetErlangServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
            case FE_HYP:
                add_event(&ev_list, GetHyperexpServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
        }
    }
}

// FS Completion Management
void FS_Completion(Event* ev) {

    // calculate average residence time in FS center. (Welford + destructive pop)
    FS_counter++;
    double res_time_FS = ev->time - arrival_pop(&arrival_queue_FS); // i-th session's
        residence time
    average_res_FS = average_res_FS + (res_time_FS - average_res_FS)/FS_counter;
    arrival_add(&arrival_queue_BES, ev->time); // save entrance time into BE center

    // Exiting from FS
    if(queue_length_FS > 0) {
        queue_length_FS--;
        switch(type) {
            case FE_EXP:
                add_event(&ev_list, GetExponentialServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
            case FE_ERL:
                add_event(&ev_list, GetErlangServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
            case FE_HYP:
                add_event(&ev_list, GetHyperexpServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
        }
    }
    else busy_FS = 0;

    // Entering in BES
    if(busy_BES) queue_length_BES++;
    else {
        busy_BES = 1;
        add_event(&ev_list, GetServiceBES(ev->time), BES_COMPL);
    }
}

// BES Completion Management
void BES_Completion(Event* ev) {

    // calculate average residence time in BES center. (Welford + destructive pop)
    BES_counter++;
    double res_time_BES = ev->time - arrival_pop(&arrival_queue_BES); // i-th session's
        residence time
    average_res_BES = average_res_BES + (res_time_BES - average_res_BES)/BES_counter;
```

```
    // Exiting from BES
    if(queue_length_BES > 0) {
        queue_length_BES--;
        add_event(&ev_list, GetServiceBES(ev->time), BES_COMPL);
    }
    else busy_BES = 0;

    //Entering Client
    unsigned int this_requests = dequeue_req(&req_queue);
    if(this_requests <= 0) {
        // session is over. It will move out of the system
        set_autocorr_data(__completed_sessions+completed_sessions,
             average_res_FS+average_res_BES);
        completed_sessions++;
    }
    else {
        // This session still has some requests to be executed. Let's go to the Clients
        requests++;
        client_counter++;
        active_client++; // increase the number of active clients

        // Calculate average residence time in Clients center
        double temp = GetServiceClient(ev->time);

        double time_res_client = temp - ev->time;
        add_event(&ev_list, temp, CL_COMPL);

        average_res_client = average_res_client + (time_res_client -
             average_res_client)/client_counter;

        // Add this time and this session's remaining requests into ClientOrderList
        add_client_req(&client_req_list, temp, this_requests-1);
    }
}

// Client Completion Management
void Client_Completion(Event* ev) {
    active_client--; // decrease number of Clients
    // Insert the updated value of remaining requests (from THIS session) into the ReqQueue
    ClientReq* coming_back_session = pop_ClientReq(&client_req_list);

    // Gestire l'abort delle sessioni
    if(threshold_flag) {
        if(threshold_exceeded) {
            if(FS_average_utilization <= THRESHOLD_MIN)
                threshold_exceeded = 0;
            else {
                aborted++;
                //sessions++; // FIXME
                free(coming_back_session);
                return;
            }
        }
        else if(FS_average_utilization >= THRESHOLD_MAX) {
            threshold_exceeded = 1;
            aborted++;
            //sessions++; // FIXME
            free(coming_back_session);
            return;
        }
    }
```

```
    enqueue_req(&req_queue, coming_back_session->req);
    free(coming_back_session);

    if(busy_FS == 0) {
        busy_FS = 1;
        switch(type) {
            case FE_EXP:
                add_event(&ev_list, GetExponentialServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
            case FE_ERL:
                add_event(&ev_list, GetErlangServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
            case FE_HYP:
                add_event(&ev_list, GetHyperexpServiceFS(ev->time), FS_COMPL); // create
                    FS_COMPL event and schedule it
                break;
        }
    }
    else queue_length_FS++;
    arrival_add(&arrival_queue_FS, ev->time);
}

void manage_event(Event *e) {
    switch(e->type) {
        case NEW_SESSION:
            NewSession(e);
            break;
        case FS_COMPL:
            FS_Completion(e);
            break;
        case BES_COMPL:
            BES_Completion(e);
            break;
        case CL_COMPL:
            Client_Completion(e);
            break;
        default:
            printf("Wrong event!\n");
    }
    free(e);
}
```

A.4.6 file_manager.c

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Gestisce tutto il flusso di dati da scrivere su file
 */

#define MAX_DATETIME 150
extern SIMULATION_TYPES type;

char* get_date() {
    time_t td = time(NULL);
  struct tm *now = NULL;
```

```
  char * buffer = (char *)malloc(sizeof(char)*(MAX_DATETIME+1));
  buffer[MAX_DATETIME] = '\0';
  now = localtime(&td);  /* Get time and date structure */
  strftime (buffer, MAX_DATETIME, "%Y%m%d-%H%M%S",now);
  return buffer;
}

FILE *open_file() {
    char graphic_name[50], *g = graphic_name, *graphic_ext = ".csv";
    char *date = get_date();
    strcpy(graphic_name, date);
    g+=strlen(date);
    switch (type) {
        case FE_EXP:
            strcat(g,"-FE_EXP");
            g+=strlen("-FE_EXP");
            break;
        case FE_ERL:
            strcat(g,"-FE_ERL");
            g+=strlen("-FE_ERL");
            break;
        case FE_HYP:
            strcat(g,"-FE_HYP");
            g+=strlen("-FE_HYP");
            break;
        default:
            printf("Error. No known simulation type.\n");
    }
    if(threshold_flag) {
        strcat(g,"_OM");
        g+=strlen("_OM");
    }
    strcat(g, graphic_ext);
    g[strlen(graphic_ext)] = 0;
    free(date);
    return fopen(graphic_name, "a");
}

void close_file(FILE *g) {
    fclose(g);
}
```

A.4.7 generate_random_value.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - genera i valori random per arrivi e servizi
 */

#define MIN_REQ 5.0
#define MAX_REQ 35.0
#define ARRIVAL_TIME 1.0/35.0  // 1/lambda
#define THINK_TIME 7.0         // E[Z]
#define FS_COMPL_TIME 0.00456  // E[D]_front tempo di servizio (quello che spende nella
     palla)
#define BES_COMPL_TIME 0.00117 // E[D]_back tempo di servizio (quello che spende nella
     palla)
```

```
#define K_ERLANG 10            // Parametro per la distribuzione della 10 Erlang
#define P_HYP 0.1              // Parametro per la sitribuzione iperesponenziale

double Exponential(double m) {
    return (-m * log(1.0 - Random()));
}

double Erlang(long n, double b) {
    long  i;
    double x = 0.0;
    for (i = 0; i < n; i++)
        x += Exponential(b);
    return (x);
}

double Hyperexponential() {
    return Exponential(2*P_HYP*FS_COMPL_TIME) + Exponential(2*(1-P_HYP)*FS_COMPL_TIME);
}

long Equilikely(long a, long b) {
    return (a + (long) (Random() * (b - a + 1)));
}

double GetArrival(double prev_time) {
    return prev_time + Exponential(ARRIVAL_TIME);
}

double GetExponentialServiceFS(double prev_time) {
    return prev_time + Exponential(FS_COMPL_TIME);
}

double GetErlangServiceFS(double prev_time) {
    return prev_time + Erlang(K_ERLANG, FS_COMPL_TIME/K_ERLANG);
}

double GetHyperexpServiceFS(double prev_time) {
    return prev_time + Hyperexponential();
}

double GetServiceBES(double prev_time) {
    return prev_time + Exponential(BES_COMPL_TIME);
}

double GetServiceClient(double prev_time) {
    return prev_time + Exponential(THINK_TIME);
}

unsigned int GetRequests() {
    return (unsigned int)Equilikely(MIN_REQ, MAX_REQ);
}
```

A.4.8 global_variables.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Variabili Globali e include
 */
```

```
#define THRESHOLD_MAX    0.85f        // Threshold massimo dopo il quale droppo
#define THRESHOLD_MIN    0.75f        // Se scendo qua sotto riprendo
#define START            0.0f         // il tempo di inizio

double throughput_sessions, throughput_requests, current_time, prev_time, FS_utilization,
     FS_average_utilization;
double __throughput_sessions, __throughput_requests, __FS_utilization,
     __FS_average_utilization, prev_batch_time_completition;
int arrivals, threshold_flag;
long opened_sessions, completed_sessions, requests, dropped, aborted;
long __opened_sessions, __completed_sessions, __requests, __dropped, __aborted;
char visual_flag;

long batch_size;         // numero di job prima di finire un batch
long batch_num;          // numero di batch prima di terminare una simulazione
long current_batch;      // a quale batch sono arrivato

#include "simulation_type.h"
#include "event_list.h"
#include "arrival_queue.h"
#include "req_queue.h"
#include "client_req.h"
#include "autocorrelation.h"
#include "interval_calculator.h"

// Definizione del tipo di simulazione
SIMULATION_TYPES type;
// Code
Event *ev_list;
Request* req_queue;        // Request Queue - it stores information regarding Requests
     amount
ClientReq* client_req_list; // Matrix with Client completion time and session requests
ArrivalNode* arrival_queue_FS; // Arrival times queue at FS
ArrivalNode* arrival_queue_BES; // Arrival times queue at BES
```

A.4.9 req_queue.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Coda delle richieste
 */


struct Request{
    unsigned int n;
    struct Request* next;
};

typedef struct Request Request;

void enqueue_req(Request** list, unsigned int t){
    if(list == NULL) {
        printf("ReqQueue is NULL\n");
        return;
    }
    Request* temp = (Request*) malloc(sizeof(Request));
```

```
    temp->n = t;
    temp->next = NULL;
    if(*list == NULL){
        *list = temp;
        return;
    }

    Request* curr = *list;
    while(curr->next != NULL){
        curr = curr->next;
    }
    curr->next = temp;
}

unsigned int dequeue_req(Request** list){
    if(list == NULL || *list ==NULL) {
        printf("ReqQueue is empty\n");
        return 0.0;
    }
    Request* curr = *list;
    *list = (*list)->next;

    unsigned int t = curr->n;
    free(curr);

    return t;
}

void print_req(Request** list){
    if(list == NULL || *list ==NULL) {
        printf("ReqQueue is empty\n");
        return;
    }
    Request *curr=*list;
    printf("[+] ReqQueue:\n");
    while(curr != NULL){
        printf("[n: %d] -- ",curr->n);
    curr = curr->next;
    }
    printf("\n");
}
```

A.4.10 rng.c

```
/* ------------------------------------------------------------------------
 * This is an ANSI C library for random number generation. The use of this
 * library is recommended as a replacement for the ANSI C rand() and srand()
 * functions, particularly in simulation applications where the statistical
 * 'goodness' of the random number generator is important.
 *
 * The generator used in this library is a so-called 'Lehmer random number
 * generator' which returns a pseudo-random number uniformly distributed
 * between 0.0 and 1.0. The period is (m - 1) where m = 2,147,483,647 and
 * the smallest and largest possible values are (1 / m) and 1 - (1 / m)
 * respectively. For more details see:
 *
 *      "Random Number Generators: Good Ones Are Hard To Find"
 *                  Steve Park and Keith Miller
```

```
 *            Communications of the ACM, October 1988
 *
 * Note that as of 7-11-90 the multiplier used in this library has changed
 * from the previous "minimal standard" 16807 to a new value of 48271. To
 * use this library in its old (16807) form change the constants MULTIPLIER
 * and CHECK as indicated in the comments.
 *
 * Name            : rng.c (Random Number Generation - Single Stream)
 * Authors         : Steve Park & Dave Geyer
 * Language        : ANSI C
 * Latest Revision : 09-11-98
 * ----------------------------------------------------------------------
 */

#include <stdio.h>
#include <time.h>
#include "rng.h"

#define MODULUS  2147483647L /* DON'T CHANGE THIS VALUE              */
#define MULTIPLIER 48271L   /* use 16807 for the "minimal standard"  */
#define CHECK    399268537L /* use 1043616065 for the "minimal standard" */
#define DEFAULT  123456789L /* initial seed, use 0 < DEFAULT < MODULUS */

static long seed = DEFAULT; /* seed is the state of the generator    */


   double Random(void)
/* ----------------------------------------------------------------------
 * Random is a Lehmer generator that returns a pseudo-random real number
 * uniformly distributed between 0.0 and 1.0. The period is (m - 1)
 * where m = 2,147,483,647 amd the smallest and largest possible values
 * are (1 / m) and 1 - (1 / m) respectively.
 * ----------------------------------------------------------------------
 */
{
  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);
}


   void PutSeed(long x)
/* ----------------------------------------------------------------------
 * Use this (optional) procedure to initialize or reset the state of
 * the random number generator according to the following conventions:
 *    if x > 0 then x is the initial seed (unless too large)
 *    if x < 0 then the initial seed is obtained from the system clock
 *    if x = 0 then the initial seed is to be supplied interactively
 * ----------------------------------------------------------------------
 */
{
  char ok = 0;

  if (x > 0L)
    x = x % MODULUS;                      /* correct if x is too large */
```

```
  if (x < 0L)
    x = ((unsigned long) time((time_t *) NULL)) % MODULUS;
  if (x == 0L)
    while (!ok) {
      printf("\nEnter a positive integer seed (9 digits or less) >> ");
      scanf("%ld", &x);
      ok = (0L < x) && (x < MODULUS);
      if (!ok)
        printf("\nInput out of range ... try again\n");
    }
  seed = x;
}


   void GetSeed(long *x)
/* ------------------------------------------------------------------
 * Use this (optional) procedure to get the current state of the random
 * number generator.
 * ------------------------------------------------------------------
 */
{
  *x = seed;
}
```

A.4.11 rng.h

```
/* ------------------------------------------------------------------------
 * Name           : rng.h (header file for the library file rng.c)
 * Author         : Steve Park & Dave Geyer
 * Language       : ANSI C
 * Latest Revision : 09-11-98
 * ------------------------------------------------------------------------
 */

#if !defined( _RNG_ )
#define _RNG_

double Random(void);
void   GetSeed(long *x);
void   PutSeed(long x);
void   TestRandom(void);

#endif
```

A.4.12 rvms.c

```
/* ------------------------------------------------------------------------
 * This is an ANSI C library that can be used to evaluate the probability
 * density functions (pdf's), cumulative distribution functions (cdf's), and
 * inverse distribution functions (idf's) for a variety of discrete and
 * continuous random variables.
 *
 * The following notational conventions are used
```

```
 *               x : possible value of the random variable
 *               u : real variable (probability) between 0.0 and 1.0
 *  a, b, n, p, m, s : distribution-specific parameters
 *
 * There are pdf's, cdf's and idf's for 6 discrete random variables
 *
 *     Random Variable  Range (x) Mean       Variance
 *
 *     Bernoulli(p)     0..1      p          p*(1-p)
 *     Binomial(n, p)   0..n      n*p        n*p*(1-p)
 *     Equilikely(a, b) a..b      (a+b)/2    ((b-a+1)*(b-a+1)-1)/12
 *     Geometric(p)     0...      p/(1-p)    p/((1-p)*(1-p))
 *     Pascal(n, p)     0...      n*p/(1-p)  n*p/((1-p)*(1-p))
 *     Poisson(m)       0...      m          m
 *
 * and for 7 continuous random variables
 *
 *     Uniform(a, b)    a < x < b (a+b)/2    (b-a)*(b-a)/12
 *     Exponential(m)   x > 0     m          m*m
 *     Erlang(n, b)     x > 0     n*b        n*b*b
 *     Normal(m, s)     all x     m          s*s
 *     Lognormal(a, b)  x > 0        see below
 *     Chisquare(n)     x > 0     n          2*n
 *     Student(n)       all x     0  (n > 1) n/(n-2)  (n > 2)
 *
 * For the Lognormal(a, b), the mean and variance are
 *
 *                     mean = Exp(a + 0.5*b*b)
 *                 variance = (Exp(b*b) - 1)*Exp(2*a + b*b)
 *
 * Name           : rvms.c (Random Variable ModelS)
 * Author         : Steve Park & Dave Geyer
 * Language       : ANSI C
 * Latest Revision : 11-22-97
 * ----------------------------------------------------------------------
 */

#include <math.h>
#include "rvms.h"

#define TINY  1.0e-10
#define SQRT2PI 2.506628274631          /* sqrt(2 * pi) */

static double pdfStandard(double x);
static double cdfStandard(double x);
static double idfStandard(double u);
static double LogGamma(double a);
static double LogBeta(double a, double b);
static double InGamma(double a, double b);
static double InBeta(double a, double b, double x);


  double pdfBernoulli(double p, long x)
/* ======================================
 * NOTE: use 0.0 < p < 1.0 and 0 <= x <= 1
 * ======================================
 */
{
  return ((x == 0) ? 1.0 - p : p);
}

  double cdfBernoulli(double p, long x)
```

```
/* ======================================
 * NOTE: use 0.0 < p < 1.0 and 0 <= x <= 1
 * ======================================
 */
{
   return ((x == 0) ? 1.0 - p : 1.0);
}

   long idfBernoulli(double p, double u)
/* ======================================
 * NOTE: use 0.0 < p < 1.0 and 0.0 < u < 1.0
 * ======================================
 */
{
   return ((u < 1.0 - p) ? 0 : 1);
}

   double pdfEquilikely(long a, long b, long x)
/* ============================================
 * NOTE: use a <= x <= b
 * ============================================
 */
{   if(x){}
   return (1.0 / (b - a + 1.0));
}

   double cdfEquilikely(long a, long b, long x)
/* ============================================
 * NOTE: use a <= x <= b
 * ============================================
 */
{
   return ((x - a + 1.0) / (b - a + 1.0));
}

   long idfEquilikely(long a, long b, double u)
/* ============================================
 * NOTE: use a <= b and 0.0 < u < 1.0
 * ============================================
 */
{
   return (a + (long) (u * (b - a + 1)));
}

   double pdfBinomial(long n, double p, long x)
/* ============================================
 * NOTE: use 0 <= x <= n and 0.0 < p < 1.0
 * ============================================
 */
{
   double s, t;

   s = LogChoose(n, x);
   t = x * log(p) + (n - x) * log(1.0 - p);
   return (exp(s + t));
}

   double cdfBinomial(long n, double p, long x)
/* ============================================
 * NOTE: use 0 <= x <= n and 0.0 < p < 1.0
 * ============================================
 */
```

```
{
  if (x < n)
    return (1.0 - InBeta(x + 1, n - x, p));
  else
    return (1.0);
}

  long idfBinomial(long n, double p, double u)
/* ================================================
 * NOTE: use 0 <= n, 0.0 < p < 1.0 and 0.0 < u < 1.0
 * ================================================
 */
{
  long x = (long) (n * p);          /* start searching at the mean */

  if (cdfBinomial(n, p, x) <= u)
    while (cdfBinomial(n, p, x) <= u)
      x++;
  else if (cdfBinomial(n, p, 0) <= u)
    while (cdfBinomial(n, p, x - 1) > u)
      x--;
  else
    x = 0;
  return (x);
}

  double pdfGeometric(double p, long x)
/* ====================================
 * NOTE: use 0.0 < p < 1.0 and x >= 0
 * ====================================
 */
{
  return ((1.0 - p) * exp(x * log(p)));
}

  double cdfGeometric(double p, long x)
/* ====================================
 * NOTE: use 0.0 < p < 1.0 and x >= 0
 * ====================================
 */
{
  return (1.0 - exp((x + 1) * log(p)));
}

  long idfGeometric(double p, double u)
/* ========================================
 * NOTE: use 0.0 < p < 1.0 and 0.0 < u < 1.0
 * ========================================
 */
{
  return ((long) (log(1.0 - u) / log(p)));
}

  double pdfPascal(long n, double p, long x)
/* ============================================
 * NOTE: use n >= 1, 0.0 < p < 1.0, and x >= 0
 * ============================================
 */
{
  double s, t;

  s = LogChoose(n + x - 1, x);
```

```
   t = x * log(p) + n * log(1.0 - p);
   return (exp(s + t));
}

   double cdfPascal(long n, double p, long x)
/* =============================================
 * NOTE: use n >= 1, 0.0 < p < 1.0, and x >= 0
 * =============================================
 */
{
   return (1.0 - InBeta(x + 1, n, p));
}

   long idfPascal(long n, double p, double u)
/* ================================================
 * NOTE: use n >= 1, 0.0 < p < 1.0, and 0.0 < u < 1.0
 * ================================================
 */
{
   long x = (long) (n * p / (1.0 - p)); /* start searching at the mean */

   if (cdfPascal(n, p, x) <= u)
     while (cdfPascal(n, p, x) <= u)
       x++;
   else if (cdfPascal(n, p, 0) <= u)
     while (cdfPascal(n, p, x - 1) > u)
       x--;
   else
     x = 0;
   return (x);
}

   double pdfPoisson(double m, long x)
/* ==================================
 * NOTE: use m > 0 and x >= 0
 * ==================================
 */
{
   double t;

   t = - m + x * log(m) - LogFactorial(x);
   return (exp(t));
}

   double cdfPoisson(double m, long x)
/* ==================================
 * NOTE: use m > 0 and x >= 0
 * ==================================
 */
{
   return (1.0 - InGamma(x + 1, m));
}

   long idfPoisson(double m, double u)
/* ==================================
 * NOTE: use m > 0 and 0.0 < u < 1.0
 * ==================================
 */
{
   long x = (long) m;                /* start searching at the mean */

   if (cdfPoisson(m, x) <= u)
```

```
    while (cdfPoisson(m, x) <= u)
      x++;
  else if (cdfPoisson(m, 0) <= u)
    while (cdfPoisson(m, x - 1) > u)
      x--;
  else
    x = 0;
  return (x);
}

  double pdfUniform(double a, double b, double x)
/* ================================================
 * NOTE: use a < x < b
 * ================================================
 */
{
   if(x) {}
  return (1.0 / (b - a));
}

  double cdfUniform(double a, double b, double x)
/* ================================================
 * NOTE: use a < x < b
 * ================================================
 */
{
  return ((x - a) / (b - a));
}

  double idfUniform(double a, double b, double u)
/* ================================================
 * NOTE: use a < b and 0.0 < u < 1.0
 * ================================================
 */
{
  return (a + (b - a) * u);
}

  double pdfExponential(double m, double x)
/* ========================================
 * NOTE: use m > 0 and x > 0
 * ========================================
 */
{
  return ((1.0 / m) * exp(- x / m));
}

  double cdfExponential(double m, double x)
/* ========================================
 * NOTE: use m > 0 and x > 0
 * ========================================
 */
{
  return (1.0 - exp(- x / m));
}

  double idfExponential(double m, double u)
/* ========================================
 * NOTE: use m > 0 and 0.0 < u < 1.0
 * ========================================
 */
{
```

```
   return (- m * log(1.0 - u));
}

   double pdfErlang(long n, double b, double x)
/* ============================================
 * NOTE: use n >= 1, b > 0, and x > 0
 * ============================================
 */
{
   double t;

   t = (n - 1) * log(x / b) - (x / b) - log(b) - LogGamma(n);
   return (exp(t));
}

   double cdfErlang(long n, double b, double x)
/* ============================================
 * NOTE: use n >= 1, b > 0, and x > 0
 * ============================================
 */
{
   return (InGamma(n, x / b));
}

   double idfErlang(long n, double b, double u)
/* ============================================
 * NOTE: use n >= 1, b > 0 and 0.0 < u < 1.0
 * ============================================
 */
{
   double t, x = n * b;               /* initialize to the mean, then */

   do {                               /* use Newton-Raphson iteration */
     t = x;
     x = t + (u - cdfErlang(n, b, t)) / pdfErlang(n, b, t);
     if (x <= 0.0)
       x = 0.5 * t;
   } while (fabs(x - t) >= TINY);
   return (x);
}

   static double pdfStandard(double x)
/* ==================================
 * NOTE: x can be any value
 * ==================================
 */
{
   return (exp(- 0.5 * x * x) / SQRT2PI);
}

   static double cdfStandard(double x)
/* ==================================
 * NOTE: x can be any value
 * ==================================
 */
{
   double t;

   t = InGamma(0.5, 0.5 * x * x);
   if (x < 0.0)
     return (0.5 * (1.0 - t));
   else
```

```
    return (0.5 * (1.0 + t));
}

  static double idfStandard(double u)
/* ==================================
 * NOTE: 0.0 < u < 1.0
 * ==================================
 */
{
  double t, x = 0.0;                /* initialize to the mean, then */

  do {                              /* use Newton-Raphson iteration */
    t = x;
    x = t + (u - cdfStandard(t)) / pdfStandard(t);
  } while (fabs(x - t) >= TINY);
  return (x);
}

  double pdfNormal(double m, double s, double x)
/* ============================================
 * NOTE: x and m can be any value, but s > 0.0
 * ============================================
 */
{
  double t = (x - m) / s;

  return (pdfStandard(t) / s);
}

  double cdfNormal(double m, double s, double x)
/* ============================================
 * NOTE: x and m can be any value, but s > 0.0
 * ============================================
 */
{
  double t = (x - m) / s;

  return (cdfStandard(t));
}

  double idfNormal(double m, double s, double u)
/* ======================================================
 * NOTE: m can be any value, but s > 0.0 and 0.0 < u < 1.0
 * ======================================================
 */
{
  return (m + s * idfStandard(u));
}

  double pdfLognormal(double a, double b, double x)
/* ================================================
 * NOTE: a can have any value, but b > 0.0 and x > 0.0
 * ================================================
 */
{
  double t = (log(x) - a) / b;

  return (pdfStandard(t) / (b * x));
}

  double cdfLognormal(double a, double b, double x)
/* ================================================
```

```
 * NOTE: a can have any value, but b > 0.0 and x > 0.0
 * ================================================
 */
{
   double t = (log(x) - a) / b;

   return (cdfStandard(t));
}

   double idfLognormal(double a, double b, double u)
/* ============================================================
 * NOTE: a can have any value, but b > 0.0 and 0.0 < u < 1.0
 * ============================================================
 */
{
   double t;

   t = a + b * idfStandard(u);
   return (exp(t));
}

   double pdfChisquare(long n, double x)
/* ====================================
 * NOTE: use n >= 1 and x > 0.0
 * ====================================
 */
{
   double t, s = n / 2.0;

   t = (s - 1.0) * log(x / 2.0) - (x / 2.0) - log(2.0) - LogGamma(s);
   return (exp(t));
}

   double cdfChisquare(long n, double x)
/* ====================================
 * NOTE: use n >= 1 and x > 0.0
 * ====================================
 */
{
   return (InGamma(n / 2.0, x / 2));
}

   double idfChisquare(long n, double u)
/* ====================================
 * NOTE: use n >= 1 and 0.0 < u < 1.0
 * ====================================
 */
{
   double t, x = n;                    /* initialize to the mean, then */

   do {                                /* use Newton-Raphson iteration */
     t = x;
     x = t + (u - cdfChisquare(n, t)) / pdfChisquare(n, t);
     if (x <= 0.0)
       x = 0.5 * t;
   } while (fabs(x - t) >= TINY);
   return (x);
}

   double pdfStudent(long n, double x)
/* ==================================
 * NOTE: use n >= 1 and x > 0.0
```

```
 * ==================================
 */
{
   double s, t;

   s = -0.5 * (n + 1) * log(1.0 + ((x * x) / (double) n));
   t = -LogBeta(0.5, n / 2.0);
   return (exp(s + t) / sqrt((double) n));
}

   double cdfStudent(long n, double x)
/* ==================================
 * NOTE: use n >= 1 and x > 0.0
 * ==================================
 */
{
   double s, t;

   t = (x * x) / (n + x * x);
   s = InBeta(0.5, n / 2.0, t);
   if (x >= 0.0)
     return (0.5 * (1.0 + s));
   else
     return (0.5 * (1.0 - s));
}

   double idfStudent(long n, double u)
/* ==================================
 * NOTE: use n >= 1 and 0.0 < u < 1.0
 * ==================================
 */
{
   double t, x = 0.0;                  /* initialize to the mean, then */

   do {                                /* use Newton-Raphson iteration */
     t = x;
     x = t + (u - cdfStudent(n, t)) / pdfStudent(n, t);
   } while (fabs(x - t) >= TINY);
   return (x);
}

/* ====================================================================
 * The six functions that follow are a 'special function' mini-library
 * used to support the evaluation of pdf, cdf and idf functions.
 * ====================================================================
 */

   static double LogGamma(double a)
/* ================================================================================
 * LogGamma returns the natural log of the gamma function.
 * NOTE: use a > 0.0
 *
 * The algorithm used to evaluate the natural log of the gamma function is
 * based on an approximation by C. Lanczos, SIAM J. Numerical Analysis, B,
 * vol 1, 1964. The constants have been selected to yield a relative error
 * which is less than 2.0e-10 for all positive values of the parameter a.
 * ================================================================================
 */
{
   double s[6], sum, temp;
   int    i;
```

```
   s[0] = 76.180091729406 / a;
   s[1] = -86.505320327112 / (a + 1.0);
   s[2] = 24.014098222230 / (a + 2.0);
   s[3] = -1.231739516140 / (a + 3.0);
   s[4] =  0.001208580030 / (a + 4.0);
   s[5] = -0.000005363820 / (a + 5.0);
   sum  =  1.000000000178;
   for (i = 0; i < 6; i++)
     sum += s[i];
   temp = (a - 0.5) * log(a + 4.5) - (a + 4.5) + log(SQRT2PI * sum);
   return (temp);
}

   double LogFactorial(long n)
/* ============================================================
 * LogFactorial(n) returns the natural log of n!
 * NOTE: use n >= 0
 *
 * The algorithm used to evaluate the natural log of n! is based on a
 * simple equation which relates the gamma and factorial functions.
 * ============================================================
 */
{
   return (LogGamma(n + 1));
}

   static double LogBeta(double a, double b)
/* ============================================================================
 * LogBeta returns the natural log of the beta function.
 * NOTE: use a > 0.0 and b > 0.0
 *
 * The algorithm used to evaluate the natural log of the beta function is
 * based on a simple equation which relates the gamma and beta functions.
 *
 */
{
   return (LogGamma(a) + LogGamma(b) - LogGamma(a + b));
}

   double LogChoose(long n, long m)
/* ============================================================================
 * LogChoose returns the natural log of the binomial coefficient C(n,m).
 * NOTE: use 0 <= m <= n
 *
 * The algorithm used to evaluate the natural log of a binomial coefficient
 * is based on a simple equation which relates the beta function to a
 * binomial coefficient.
 * ============================================================================
 */
{
   if (m > 0)
     return (-LogBeta(m, n - m + 1) - log(m));
   else
     return (0.0);
}

   static double InGamma(double a, double x)
/* ============================================================================
 * Evaluates the incomplete gamma function.
 * NOTE: use a > 0.0 and x >= 0.0
 *
 * The algorithm used to evaluate the incomplete gamma function is based on
```

```
 * Algorithm AS 32, J. Applied Statistics, 1970, by G. P. Bhattacharjee.
 * See also equations 6.5.29 and 6.5.31 in the Handbook of Mathematical
 * Functions, Abramowitz and Stegum (editors). The absolute error is less
 * than 1e-10 for all non-negative values of x.
 * ======================================================================
 */
{
   double t, sum, term, factor, f, g, c[2], p[3], q[3];
   long  n;

   if (x > 0.0)
     factor = exp(-x + a * log(x) - LogGamma(a));
   else
     factor = 0.0;
   if (x < a + 1.0) {             /* evaluate as an infinite series - */
     t    = a;                    /* A & S equation 6.5.29         */
     term = 1.0 / a;
     sum  = term;
     while (term >= TINY * sum) {  /* sum until 'term' is small */
       t++;
       term *= x / t;
       sum  += term;
     }
     return (factor * sum);
   }
   else {                         /* evaluate as a continued fraction - */
     p[0] = 0.0;                  /* A & S eqn 6.5.31 with the extended */
     q[0] = 1.0;                  /* pattern 2-a, 2, 3-a, 3, 4-a, 4,... */
     p[1] = 1.0;                  /* - see also A & S sec 3.10, eqn (3) */
     q[1] = x;
     f    = p[1] / q[1];
     n    = 0;
     do {                         /* recursively generate the continued */
       g  = f;                    /* fraction 'f' until two consecutive */
       n++;                       /* values are small               */
       if ((n % 2) > 0) {
         c[0] = ((double) (n + 1) / 2) - a;
         c[1] = 1.0;
       }
       else {
         c[0] = (double) n / 2;
         c[1] = x;
       }
       p[2] = c[1] * p[1] + c[0] * p[0];
       q[2] = c[1] * q[1] + c[0] * q[0];
       if (q[2] != 0.0) {         /* rescale to avoid overflow */
         p[0] = p[1] / q[2];
         q[0] = q[1] / q[2];
         p[1] = p[2] / q[2];
         q[1] = 1.0;
         f    = p[1];
       }
     } while ((fabs(f - g) >= TINY) || (q[1] != 1.0));
     return (1.0 - factor * f);
   }
}

   static double InBeta(double a, double b, double x)
/* ======================================================================
 * Evaluates the incomplete beta function.
 * NOTE: use a > 0.0, b > 0.0 and 0.0 <= x <= 1.0
 *
```

```
 * The algorithm used to evaluate the incomplete beta function is based on
 * equation 26.5.8 in the Handbook of Mathematical Functions, Abramowitz
 * and Stegum (editors). The absolute error is less than 1e-10 for all x
 * between 0 and 1.
 * ======================================================================
 */
{
   double t, factor, f, g, c, p[3], q[3];
   int    swap;
   long   n;

   if (x > (a + 1.0) / (a + b + 1.0)) { /* to accelerate convergence */
     swap = 1;                        /* complement x and swap a & b */
     x    = 1.0 - x;
     t    = a;
     a    = b;
     b    = t;
   }
   else                               /* do nothing */
     swap = 0;
   if (x > 0)
     factor = exp(a * log(x) + b * log(1.0 - x) - LogBeta(a,b)) / a;
   else
     factor = 0.0;
   p[0] = 0.0;
   q[0] = 1.0;
   p[1] = 1.0;
   q[1] = 1.0;
   f    = p[1] / q[1];
   n    = 0;
   do {                             /* recursively generate the continued */
     g = f;                         /* fraction 'f' until two consecutive */
     n++;                           /* values are small                   */
     if ((n % 2) > 0) {
       t = (double) (n - 1) / 2;
       c = -(a + t) * (a + b + t) * x / ((a + n - 1.0) * (a + n));
     }
     else {
       t = (double) n / 2;
       c = t * (b - t) * x / ((a + n - 1.0) * (a + n));
     }
     p[2] = p[1] + c * p[0];
     q[2] = q[1] + c * q[0];
     if (q[2] != 0.0) {             /* rescale to avoid overflow */
       p[0] = p[1] / q[2];
       q[0] = q[1] / q[2];
       p[1] = p[2] / q[2];
       q[1] = 1.0;
       f    = p[1];
     }
   } while ((fabs(f - g) >= TINY) || (q[1] != 1.0));
   if (swap)
     return (1.0 - factor * f);
   else
     return (factor * f);
}
```

A.4.13 rvms.h

```
/* ------------------------------------------------------------
 * Name           : rvms.h (header file for the library rvms.c)
 * Author         : Steve Park & Dave Geyer
 * Language       : ANSI C
 * Latest Revision : 11-02-96
 * ------------------------------------------------------------
 */

#if !defined( _RVMS_ )
#define _RVMS_

double LogFactorial(long n);
double LogChoose(long n, long m);

double pdfBernoulli(double p, long x);
double cdfBernoulli(double p, long x);
long   idfBernoulli(double p, double u);

double pdfEquilikely(long a, long b, long x);
double cdfEquilikely(long a, long b, long x);
long   idfEquilikely(long a, long b, double u);

double pdfBinomial(long n, double p, long x);
double cdfBinomial(long n, double p, long x);
long   idfBinomial(long n, double p, double u);

double pdfGeometric(double p, long x);
double cdfGeometric(double p, long x);
long   idfGeometric(double p, double u);

double pdfPascal(long n, double p, long x);
double cdfPascal(long n, double p, long x);
long   idfPascal(long n, double p, double u);

double pdfPoisson(double m, long x);
double cdfPoisson(double m, long x);
long   idfPoisson(double m, double u);

double pdfUniform(double a, double b, double x);
double cdfUniform(double a, double b, double x);
double idfUniform(double a, double b, double u);

double pdfExponential(double m, double x);
double cdfExponential(double m, double x);
double idfExponential(double m, double u);

double pdfErlang(long n, double b, double x);
double cdfErlang(long n, double b, double x);
double idfErlang(long n, double b, double u);

double pdfNormal(double m, double s, double x);
double cdfNormal(double m, double s, double x);
double idfNormal(double m, double s, double u);

double pdfLognormal(double a, double b, double x);
double cdfLognormal(double a, double b, double x);
double idfLognormal(double a, double b, double u);

double pdfChisquare(long n, double x);
double cdfChisquare(long n, double x);
double idfChisquare(long n, double u);
```

```
double pdfStudent(long n, double x);
double cdfStudent(long n, double x);
double idfStudent(long n, double u);

#endif
```

A.4.14 simulation_type.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Struttura dati per la simulazione
 */

enum _SIMULATION_TYPES {
    FE_EXP,
    FE_ERL,
    FE_HYP
};

typedef enum _SIMULATION_TYPES SIMULATION_TYPES;

char* simulation_traslator(SIMULATION_TYPES t){
  switch(t){
    case FE_EXP:
            return "Exponential";
    break;
    case FE_ERL:
            return "Erlang";
    break;
    case FE_HYP:
            return "Hyperexponential";
    break;
    default:
            return "Error event";
    break;
  }
  return "Wrong code!\n";
}
```

A.4.15 user_signal.c

```
#include <signal.h>

void sigprint() {
    clear_console();
    print_system_state(SYSTEM_PRINT);
}

void sigauto() {
    clear_console();
    print_system_state(SYSTEM_PRINT);
    exit(EXIT_SUCCESS);
```

```
}

void add_signals() {
    signal(SIGUSR1, sigprint);
    signal(SIGUSR2, sigauto);
}
```

A.4.16 utils.h

```
/**
 *      Progetto MPSR anno 14/15.
 *      Authori: S. Martucci, A. Valenti
 *      Simulatore web - Utility varie per la stampa
 */
void clear_console() {
#ifdef __WIN32
    system("cls");
#else
    // Assume POSIX
    char str[] = {0x1b, 0x5b, 0x48, 0x1b, 0x5b, 0x4a, '\0'};
    printf("%s", str);;
#endif
}

void print_initial_settings(FILE *g, long long int seed, long bs, long bn) {
    fprintf(g, "%s\t%lld\t\t%s\t%s\n", "SEED", seed, "FS distribution",
        simulation_traslator(type));
    fprintf(g, "%s\t%ld\t\t%s\t%ld\n", "Batch Size", bs, "Total Batch", bn);
    fprintf(g, "%s\t%s\t\n\n", "Threshold", (threshold_flag) ? "YES : 85%" : "NONE");
    fprintf(g, "%s\t%s\t%s\t%s\t", "Current batch", "Util FS", "Sessions", "Requests");
    fprintf(g, "%s\t%s\t%s\t%s\t", "Average Response Time", "Throughput (Sessions)",
        "Throughput (Requests)", "Average Request per Session");
    fprintf(g, "%s\t%s\t%s\t%s\t%s\t%s\t\n\n", "# Completed", "% Completed", "# Dropped",
        "% Dropped", "# Aborted", "% Aborted");
    fflush(g);
}

void print_system_state_on_file(FILE *g) {
    fprintf(g, "%ld\t%6.16f\t%ld\t%ld\t", current_batch, FS_average_utilization,
        opened_sessions, requests);
    fprintf(g, "%6.8f\t%6.8f\t%6.8f\t%6.8f\t", average_res_FS + average_res_BES,
        throughput_sessions, throughput_requests, (double) requests/opened_sessions);
    fprintf(g, "%ld\t%6.8f\t%ld\t%6.8f\t%ld\t%6.8f\t\t\n", completed_sessions, ((double)
        completed_sessions/opened_sessions)*100.0, dropped,
        ((double)dropped/(dropped+opened_sessions))*100.0, aborted,
        ((double)aborted/opened_sessions)*100.0);
    fflush(g);
}

void print_final_state(FILE *g) {
    fprintf(g, "\n");
    fprintf(g, "%s\t%6.16f\t%ld\t%ld\t", "Final Statistics",
        __FS_average_utilization/batch_num, __opened_sessions/batch_num,
        __requests/batch_num);
    fprintf(g, "%6.8f\t%6.8f\t%6.8f\t%6.8f\t", average_res_FS + average_res_BES,
        __throughput_sessions, __throughput_requests,
        (double)__requests/__opened_sessions);
```

```
    fprintf(g, "%ld\t%6.8f\t%ld\t%6.8f\t%ld\t%6.8f\t\t\n", __completed_sessions,
        ((double)__completed_sessions/__opened_sessions)*100.0, __dropped,
        ((double)__dropped/(__dropped+__opened_sessions))*100.0, __aborted,
        ((double)__aborted/__opened_sessions)*100.0);
    fprintf(g, "%s\t%6.8f\n", "ELAPSED TIME:", current_time);
    fflush(g);
}

void print_system_state(EVENT_TYPE t) {
    printf("::::::::::::::::::::::::::::::: Timer :::::::::::::::::::::::::::::::\n");
    printf("Current batch.............: %ld of %ld\n", current_batch, batch_num);
    printf("Current job...............: %ld of %ld\n", completed_sessions, batch_size);
    printf("Current time..............: %6.8f\n", current_time);
    printf("\n");
    printf("::::::::::::::::::::::::: Front Server Info :::::::::::::::::::::::::\n");
    printf("Choosen distribution......: %s\n", simulation_traslator(type));
    printf("Queue length..............: %d\n", queue_length_FS);
    printf("Active....................: %s\n", (busy_FS) ? "YES" : "NO");
    printf("Average response time.....: %6.8f\n", average_res_FS);
    printf("Average utilization.......: %6.8f%%\n", FS_average_utilization*100);
    printf("\n");
    printf("::::::::::::::::::::::: Back-end Server Info ::::::::::::::::::::::\n");
    printf("Queue length..............: %d\n", queue_length_BES);
    printf("Active....................: %s\n", (busy_BES) ? "YES" : "NO");
    printf("Average response time.....: %6.8f\n", average_res_BES);
    printf("\n");
    printf("::::::::::::::::::::::::::: Client Info ::::::::::::::::::::::::::\n");
    printf("Number of active client...: %d\n", active_client);
    printf("Average response time.....: %6.8f\n", average_res_client);
    printf("\n");
    printf("::::::::::::::::::::::::::: Generic Info ::::::::::::::::::::::::::\n");
    printf("Event type................: %s\n", event_translator(t));
    printf("Opened sessions...........: %ld\n", opened_sessions);
    printf("Completed sessions........: %ld\n", completed_sessions);
    printf("Completed requests........: %ld\n", requests);
    printf("Requests per session......: %6.8f\n", (double) requests/opened_sessions);
    printf("Sessions throughput.......: %6.8f\n", throughput_sessions);
    printf("Requests throughput.......: %6.8f\n", throughput_requests);
    printf("Connection dropped........: %ld\n", dropped);
    printf("Connection aborted........: %ld\n", aborted);
    printf("Threshold.................: %s\n", threshold_flag ? "YES : 85%" : "NONE");
    printf("\n");
}
```

Appendice B

Grafici

B.1 Distribuzione 10 Erlang

B.2 Distribuzione Iperesponenziale

B.3 Grafici delle autocorrelazioni

B.4 Distribuzione 10 Erlang con Overload Manager

B.5 Distribuzione Esponenziale

B.6 Grafici delle autocorrelazioni