Università Ca' Foscari - Venezia
Corso di Laurea Specialistica in Informatica

DOCUMENTAZIONE DEL PROGETTO DI PRESTAZIONI E AFFIDABILITÀ DI SISTEMI
AA 2006/2007

Possamai Lino, 800509
lino@possamai.it

Indice

Capitolo 1

Primo Compito

1.1 Esercizio 1

Questo primo esercizio consiste nel considerare un sistema rappresentabile da una catena di Marcov con matrice di probabilità di transizione (quindi una catena a tempo discreto) contenente solo 0 e 1 e nel descrivere le varie possibilità di comportamento della catena.

Introduciamo brevemente la notazione utilizzata per la classificazione degli stati. Una catena di Marcov si dice **irriducibile** se da ogni nodo è possibile raggiungere ogni altro nodo. Sia $f_j^{(k)}$ la probabilità che si ritorni allo stato E_j dopo k passi (partendo da E_j). Sia inoltre $f_j = \sum_{n=1}^{\infty} f_j^{(n)}$ la probabilità di ritornare prima o poi a E_j e $M_j = \sum_{n=1}^{\infty} n f_j^{(n)}$ il tempo medio di ricorrenza allo stato E_j.

A questo punto possiamo definire lo stato E_j come:

- **Ricorrente**, se $f_j = 1$,
  - **Ricorrente non nullo**, se $M_j < \infty$.
  - R**icorrente nullo**, se $M = \infty$.
- **Transitorio**, se $f_j < 1$,
- **Periodico** di periodo γ se il numero di passi in cui si può tornare a E_j è $\gamma, 2\gamma, 3\gamma, ...$
- **Aperiodico**, se $\gamma = 1$.

Per ora consideriamo catene discrete a stati finiti (con cardinalità pari a n). In ogni riga della matrice delle transizioni c'è un 1 in qualche posizione e 0 in tutte le altre (deve valere $\sum_j p_{ij} = 1$). Cerchiamo di individuare i possibili schemi di assegnazione degli 1.

Matrice identità I. Rappresenta una catena di Marcov non irriducibile formata da n sottocatene chiuse formate ciascuna da un solo elemento. Da questo segue che $f_j^{(1)} = 1 \ \forall j$ per cui $\sum_{k=1}^{\infty} f_j^{(k)} = 1$. Tutti gli stati saranno ricorrenti non nulli perché ad ogni passo si ritorna allo stato di partenza. Gli stati non sono né transitori né periodici.

Catena ciclica. È una catena irriducibile in cui $p_{ii+1} = 1$, $\forall i < n$ e $p_{n1} = 1$. Graficamente possiamo rappresentare questa tipologia di catena con la seguente matrice (ipotizzando $n = 5$):

$$p_{ij} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

e dal diagramma degli stati di figura 1.1.

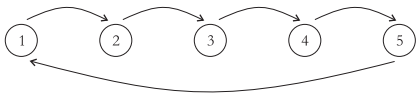


Figura 1.1: Esempio di grafo che rappresenta una catena di Marcov ciclica.

È possibile descrivere le catene di Marcov con un grafo orientato in cui il grado di uscita di ogni nodo è pari a 1 come anche il grado in entrata.

Rispetto ai vincoli sulla matrice presentati in precedenza, possiamo notare come sia possibile creare componenti irriducibili chiuse a cui sono collegati dei cammini che convogliano gli utenti verso una sottocatena chiusa (vedi per esempio la figura 1.2).

In generale è possibile descrivere le possibili situazioni che si possono creare specificando i seguenti parametri:

- c rappresenta il numero di componenti chiuse della catena di Marcov,
- cc_i rappresenta la cardinalità della componente chiusa i-esima.
- lt_{ij} rappresenta la lunghezza massima del cammino in ingresso al nodo i-esimo della componente chiusa j-esima.

Ogni sottocatena chiusa è irriducibile e partendo da un qualsiasi nodo (stato) di questa ci si può ritornare dopo esattamente c step. Inoltre, gli stati sono ricorrenti non nulli dato che vale $f_j = 1$ e aperiodici. Tutti gli altri stati che portano alle componenti irriducibili chiuse sono transitori, aperiodici e non ricorrenti.

1.2 Esercizio 2

In questo esercizio si considera un sistema rappresentabile dalla coda M/M/m/B con m$\geq$ 1 (numero di serventi a disposizione) e B$\geq$m (capacità del sistema) e si calcolano gli indici di prestazione quali utilizzazione, numero medio di utenti in coda e nel sistema, probabilità che un utente debba aspettare in coda e condizione di stabilità. Si suppone inoltre che i tempi di interarrivo siano esponenziali come pure i tempi di servizio.

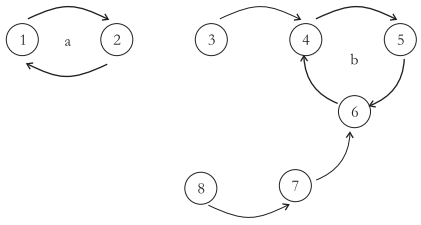


Figura 1.2: Esempio di diagramma degli stati. Si noti la presenza di due componenti irriducibili ($c = 2$) la cui cardinalità è rispettivamente pari a 2 e 3. Sono inoltre presenti due cammini che portano alla sottocatena ciclica b. Tutti gli archi rappresentano probabilità di transizione uguale a 1.

Prima di tutto sottolineiamo che in base ai valori che si assegnano a m e B, si ottengono code con caratteristiche diverse e quindi con indici di prestazione differenti. Proviamo ad analizzarli uno ad uno in dettaglio. Il valore della capacità del sistema (coda più serventi) può essere descritto in questo modo: $B = m + c$ dove m rappresenta il numero di utenti che possono essere in servizio contemporaneamente mentre c indica il numero di utenti che possono essere in coda. Nella figura 1.3 sono rappresentati tutti i vari casi che si possono presentare.

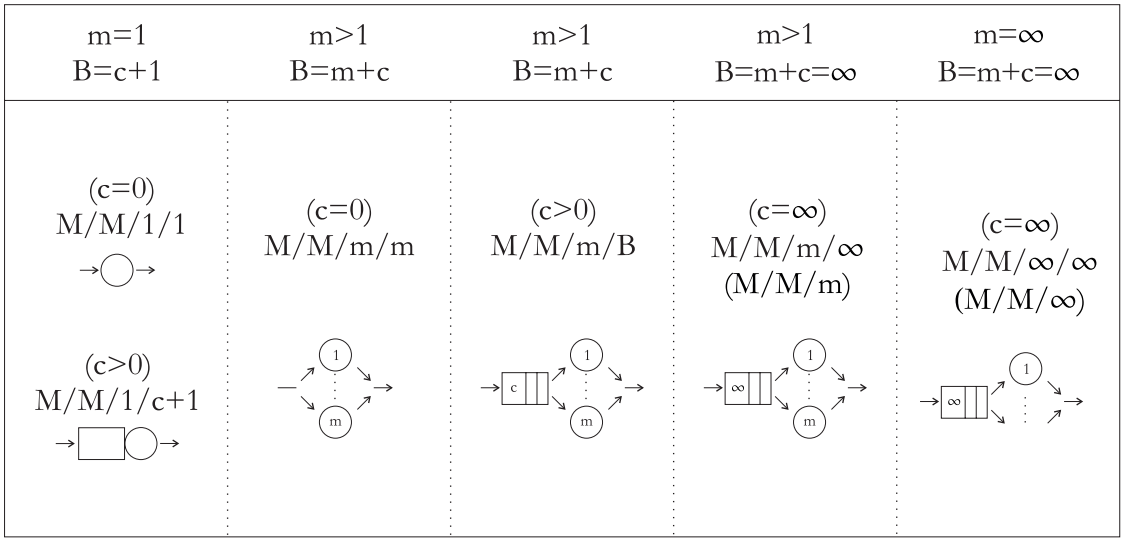


Figura 1.3: Schema riepilogativo delle tipologie di code che si possono delineare con m≥1 e B≥m.

1.2.1 M/M/1/c+1

Il sistema che considereremo in questa sezione è formata da un servente e da una coda di lunghezza c. Come tutti i sistemi a capacità finita, raggiunto il numero massimo di clienti, il sistema ne rifiuterà l'ingresso di nuovi. Il sistema può essere quindi rappresentato da un processo di nascita e morte con i

seguenti tassi:

$$\lambda_k = \begin{cases} \lambda & \text{se } k < B \\ 0 & \text{altrimenti} \end{cases} \qquad \mu_k = \mu \text{ con } k = 1, ..., B$$

Il sistema sarà sempre stabile perché nel sistema non sono ammessi più di $c + 1$ clienti. L'utilizzazione del sistema è ovviamente pari a $\rho = \frac{\lambda}{\mu}$.

Per sapere il numero medio di utenti nel sistema dobbiamo prima calcolare la probabilità che nel sistema ci siano k utenti, cioè:

$$p_k = \begin{cases} \frac{1-\rho}{1-\rho^{B+1}}\rho^k & \text{se } 0 \leq k \leq B \\ 0 & \text{altrimenti} \end{cases}$$

da cui segue che il numero medio di utenti nel sistema è pari a:

$$\begin{aligned} N &= \sum_{k=0}^{\infty} kp^k \\ &= \sum_{k=0}^{B} kp^k = \frac{1-\rho}{1-\rho^{B+1}} \sum_{k=0}^{B} k\rho^k \\ &= \frac{\rho(\rho^B(B\rho - B - 1) + 1)}{(\rho - 1)(\rho^{B+1} - 1)} \\ &= \frac{1-\rho}{1-\rho^{B+1}} \cdot \frac{\rho^{B+1}(B\rho - B - 1) + \rho}{(\rho - 1)^2} \\ &= \frac{\rho}{1-\rho} - \frac{B+1}{1-\rho^{B+1}} \cdot \rho^{B+1} \end{aligned} \tag{1.1}$$

Per calcolare il response time R, usiamo la legge di Little $N = X \cdot R$, per cui $R = N/X$ e sapendo che $\rho = \lambda/\mu$ allora $\lambda = \rho \cdot \mu$ che sostituita in 1.1 si ottiene:

$$R = \frac{N}{\lambda} \tag{1.2}$$

$$= \frac{\rho^B(B\rho - B - 1) + 1}{(\rho - 1)(\rho^{B+1} - 1)\mu} \tag{1.3}$$

per cui il numero medio di utenti in coda saranno pari a:

$$\begin{aligned} Q &= \lambda(R - 1/\mu) = N - \rho \\ &= \frac{\rho(\rho^B(B\rho - B - \rho^2 + \rho - 1) + \rho)}{(\rho - 1)(\rho^{B+1} - 1)} \end{aligned}$$

Infine, calcoliamo la probabilità che un utente debba aspettare in coda, cioè la probabilità che nel sistema ci siano 1,2, ..., c utenti, cioè

$$\begin{aligned} \sum_{k=1}^{c} p_k &= \frac{1-\rho}{1-\rho^{c+2}} \sum_{k=1}^{c} \rho^k \\ &= \frac{\rho - \rho^{c+1}}{1-\rho^{c+2}} \end{aligned}$$

1.2.2 M/M/1/1

Questa situazione si presenta quando $m = 1$ e $c = 0$. È un sistema con un servente e senza coda (la possiamo vedere come la versione base del sistema presentato nella sezione precedente). Se un cliente trova il servente occupato, è costretto a lasciare il sistema. In letteratura, questo sistema è denominato *sistema a eliminazione delle chiamate bloccate*. Il comportamento può essere modellato con un processo di nascita e morte con i seguenti tassi di arrivo e di servizio:

$$\lambda_k = \begin{cases} \lambda & \text{se } k = 0 \\ 0 & \text{altrimenti} \end{cases} \qquad \mu_k = \begin{cases} \mu & \text{se } k = 1 \\ 0 & \text{altrimenti} \end{cases}$$

e dalla probabilità p_k della coda analizzata alla sezione precedente, istanziata opportunamente, otteniamo:

$$p_k = \begin{cases} \frac{1}{1+\rho} & \text{se } k = 0 \\ \frac{\rho}{1+\rho} & \text{se } k = 1 \\ 0 & \text{altrimenti} \end{cases}$$

Anche qui, il sistema sarà sempre stabile perché tutti i clienti che trovano il servente occupato lasciano il sistema. La probabilità che un cliente debba rimanere in coda è zero perché non esiste coda. Il numero medio di utenti nel sistema e in coda sono pari a:

$$N = \sum_{k=0}^{1} k \cdot p_k = \frac{\rho}{1+\rho}, \qquad Q = 0$$

Infine, l'utilizzazione è pari alla probabilità che il sistema abbia 1 utente, cioè: $\rho = 1 - p_0 = \frac{\lambda/\mu}{1+\lambda/\mu}$.

1.2.3 M/M/m/m

Consideriamo ora il caso in cui $m > 1$, $B = m + c$ e $c = 0$. Il sistema corrispondente sarà un M/M/m/m, m serventi e con capacità finita pari a m. Il sistema non ha coda e gestisce un numero massimo di clienti pari al numero di serventi assegnandone uno ad ogni arrivo accettato, altrimenti il cliente lascia il sistema (ed è per questo che nuovamente c'è stabilità). La probabilità che un utente si trovi in coda è zero perché non esiste coda ed il numero medio di utenti in coda è sempre zero.

La probabilità che il sistema abbia k utenti sarà:

$$p_k = \begin{cases} p_0 \cdot \frac{\rho^k}{k!} & \text{se } k \leq m \\ 0 & \text{altrimenti} \end{cases} \qquad p_0 = \left[\sum_{k=0}^{m} \rho^k \cdot \frac{1}{k!} \right]^{-1}$$

con $\rho = \lambda/\mu$, per cui il numero medio di utenti nel sistema sarà:

$$N = \sum_{k=0}^{m} k \cdot p_k = \sum_{k=1}^{m} k \cdot p_k = p_0 \sum_{k=1}^{m} \frac{\rho^k}{(k-1)!}$$

L'utilizzazione corrisponde alla frazione del tempo in cui m serventi sono occupati, cioè:

$$p_m = p_0 \cdot \frac{\rho^m}{m!}$$

1.2.4 M/M/m/B

In questo sistema, ci sono m serventi ed al massimo B utenti, con $B > m$ e $c > 0$. Possiamo formulare il sistema in termini di un processo di nascita e morte con i seguenti tassi:

$$\lambda_k = \begin{cases} \lambda & \text{se } 0 \leq k < B \\ 0 & \text{altrimenti} \end{cases} \qquad \mu_k = \begin{cases} k\mu & \text{se } 0 \leq k \leq m \\ m\mu & \text{se } m \leq k \leq B \end{cases}$$

Il sistema è stabile se vale la condizione $\rho = \frac{\lambda}{m\mu} < 1$ (ed è anche il valore dell'utilizzazione). La probabilità che nel sistema ci siano k utenti sarà:

$$p_k = \begin{cases} p_0 \frac{(m\rho)^k}{k!} & \text{se } 0 \leq k \leq m \\ p_0 \frac{\rho^k \cdot m^m}{m!} & \text{se } m < k \leq B \\ 0 & \text{altrimenti} \end{cases} \qquad p_0 = \left[\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{m^m}{m!} \cdot \frac{\rho^m - \rho^{B+1}}{1-\rho} \right]^{-1}$$

da cui segue il numero medio di utenti nel sistema:

$$\begin{aligned} N &= \sum_{k=0}^{\infty} k p_k = \sum_{k=0}^{m} p_0 \frac{(m\rho)^k}{k!} + \sum_{k=m+1}^{B} p_0 \frac{(m^m \rho^k)^k}{m!} \\ &= p_0 \sum_{k=0}^{m} \frac{(m\rho)^k}{k!} + p_0 \frac{m^m}{m!} \sum_{k=m+1}^{B} \rho^k \\ &= p_0 \sum_{k=0}^{m} \frac{(m\rho)^k}{k!} + p_0 \frac{m^m}{m!} \cdot \frac{\rho^{m+1} - \rho^{B+1}}{1-\rho} \end{aligned}$$

ed il numero medio di utenti in coda (usando la legge di Little):

$$Q = m\rho\mu(R - 1/\mu) = m\rho\mu \left(\frac{N}{m\rho\mu} - 1/\mu \right) = N - m\rho$$

Infine, la probabilità che un utente debba aspettare in coda è uguale a:

$$\begin{aligned} \sum_{k=m}^{B-1} p_k &= p_0 \frac{m^m}{m!} \sum_{k=m+1}^{B-1} \rho^k + \frac{(m\rho)^m p_0}{m!} \\ &= p_0 \frac{m^m}{m!} \left(\sum_{k=m+1}^{B-1} \rho^k + \rho^m \right) \\ &= p_0 \frac{m^m}{m!} \left(\frac{\rho^B}{\rho - 1} + \frac{\rho^{m+1}}{1-\rho} + \rho^m \right) \\ &= p_0 \frac{m^m}{m!} \cdot \frac{\rho^B - \rho^m}{\rho - 1} \end{aligned}$$

1.2.5 M/M/m/∞ e M/M/∞/∞

In quest'ultima sezione si prenderanno in considerazione i casi in cui $m > 1$, $B = \infty$ e $m = \infty$, $B = \infty$. Come si può notare immediatamente, questi due sistemi a coda corrispondono a sistemi già trattati ampiamente in letteratura e precisamente corrispondono a code M/M/m e M/M/∞. Riassumiamo brevemente

i risultati. Per quanto riguarda il sistema con m servitori e infinita capacità, è possibile rappresentarlo come un processo di nascita e morte con i seguenti tassi:

$$\lambda_k = \lambda \text{ con } k = 0, 1, ... \qquad \mu_k = \begin{cases} k\mu & \text{se } 0 \leq k \leq m \\ m\mu & \text{se } k > m \end{cases}$$

Ovviamente il sistema è stabile se vale la seguente condizione: $\rho = \lambda/m\mu < 1$ (l'equazione corrisponde anche all'utilizzazione del sistema). La probabilità che nel sistema ci siano k utenti è:

$$p_k = \begin{cases} p_0 \cdot \frac{(m\rho)^k}{k!} & \text{se } k \leq m \\ p_0 \cdot \frac{m^m \rho^k}{m!} & \text{se } k > m \end{cases} \qquad \text{con } p_0 = \left[\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \left(\frac{(m\rho)^m}{m!} \right) \left(\frac{1}{1-\rho} \right) \right]^{-1}$$

Il numero medio di utenti nel sistema ed in coda è

$$N = m \cdot \rho + \frac{\rho}{(1-\rho)} \cdot \frac{(m\rho)^m}{m!(1-\rho)} \cdot p_0, \qquad Q = \frac{\rho}{(1-\rho)} \cdot \frac{(m\rho)^m}{m!(1-\rho)} \cdot p_0$$

Infine, la probabilità che un utente debba aspettare in coda è:

$$\sum_{k=m}^{\infty} p_k = \frac{(m\rho)^m}{m!(1-\rho)} \cdot p_0$$

Nel secondo caso, cioè quando abbiamo a disposizione un sistema con servitori immediatamente disponibili, è possibile utilizzare ancora un processo di nascita e morte avente i seguenti tassi:

$$\lambda_k = \lambda, \ k = 0, 1, 2, ... \qquad \mu_k = k\mu, \ k = 1, 2, ...$$

Il sistema è sempre stabile dato che ad ogni utente in arrivo esiste un servente disponibile per cui vale anche che la probabilità di attesa in coda è uguale a zero. L'utilizzazione nei sistemi di questo tipo corrisponde al numero medio di utenti nel sistema ed è pari a:

$$N = \frac{\lambda}{\mu}$$

mentre il numero medio di utenti in coda è ovviamente zero.

1.3 Esercizio 3

L'esercizio consiste nel creare un modello analitico di un sistema che rappresenta un sito web per il commercio elettronico al quale arrivano clienti che possono visionare e/o acquistare dei prodotti. In [6] si afferma che il successo nell'utilizzare un modello che rappresenta un sistema consiste proprio nel tralasciare molti dettagli di basso livello i quali sono ininfluenti molto spesso per il calcolo delle performance di alto livello, per cui anche noi utilizzeremo questo approccio top-down.

Vediamo brevemente quali sono le componenti che lo costituiscono. Il sistema è formato da tre "nodi" di cui il primo ed il terzo sono usati per astrarre il webserver che visualizza i prodotti (primo nodo) rispettivamente e che gestisce la transazione economica (ultimo nodo) mentre il secondo è utilizzato come

astrazione del fatto che gli utenti dopo aver visionato uno o più prodotti valutano (pensano) l'azione successiva da fare (acquistare il prodotto, visionarne un altro oppure uscire dal sito). Entrando più in dettaglio, il primo è composto da un servente unico, coda infinita (disciplina FIFO per assunzione), tempo di servizio esponenziale di media $S_1 = 1/\mu_1$ msec. Sempre a questo nodo entra un flusso esterno di Poisson con tasso λ. Dopo aver preso servizio al nodo 1, i clienti pensano (think time Z=$S_2 = 1/\mu_2$) e con probabilità di transizione p_{21} ritornano al servente 1 per visionare un altro prodotto mentre con probabilità p_{23} accettano di acquistare il prodotto visto e vengono trasferiti al nodo successivo per terminare la transazione economica. Il secondo nodo viene modellato come un delay server formato da infiniti serventi e nessuna coda. Dal questo nodo, gli utenti, con probabilità $p_{20} = 1 - (p_{21} + p_{23})$, escono dal sistema senza acquistare nessun prodotto. Il servente sicuro ha tempo di servizio esponenziale di media $S_3 = 1/\mu_3$ msec e coda infinita (sempre con disciplina FIFO per assunzione). Sia il primo che l'ultimo nodo descritto possono essere modellati come processi di nascita e morte con tasso di nascita λ_i, i=1, 3 e morte μ_i, i=1, 3. Se si assume che $\lambda_i < \mu_i$ allora la CTMC[1] sottostante è ergotica per cui il sistema a coda è stabile. La rappresentazione grafica del modello del sistema è in figura 1.4.

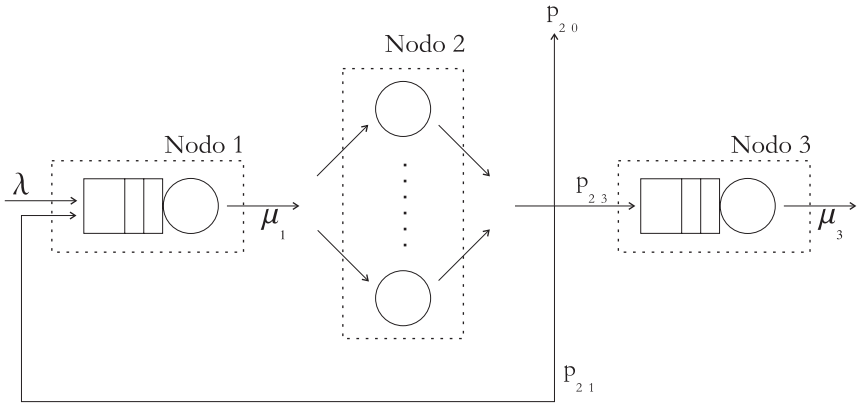


Figura 1.4: Modello a rete di code del sito di commercio elettronico.

1.3.1 Modello Analitico del sistema

Dato che gli utenti arrivano dall'esterno al nodo 1 ed escono dai nodi 2 e 3, classifichiamo la rete appena descritta come una rete aperta in cui il numero di clienti nel sistema varierà in base al tasso di arrivo λ. I nodi della rete di code possono essere analizzati singolarmente ed essere rappresentati rispettivamente da un sistema a code M/M/1 per quanto riguarda il nodo 1, da un sistema M/M/∞ (perché corrisponde ad un delay server) per il nodo 2 e nuovamente da un M/M/1 per il nodo 3. Ora, per ognuno, utilizzando la teoria delle code, si possono ricavare gli indici necessari allo studio del comportamento del sito di commercio elettronico.

Facendo riferimento alla letteratura dei sistemi a coda, saremo tentati ad utilizzare le reti di Jackson, per l'analisi di reti aperte. Da un'analisi attenta delle richieste di applicabilità del teorema è emerso che

[1] Continuous-time Markov chain

non tutte sono rispettate (precisamente quella che richiede che la disciplina di servizio di ogni servente sia FIFO). Infatti, nel nostro sistema, il nodo 2 è un servente senza coda. A questo punto, siamo obbligati a considerare la naturale estensione delle reti di Jackson, cioè le reti BCMP. Grazie a queste, si possono modellare moltissime situazioni, tra le quali la nostra che comprende un delay server.

Ripercorriamo ora tutti gli step necessari al calcolo degli indici di prestazione. Prima di tutto, bisogna calcolare i visit ratio e_i di ogni nodo che per reti aperte si calcolano come:

$$e_{ir} = p_{0,ir} + \sum_{j=1}^{3} e_{js} \cdot p_{js,ir}$$

con $s \in C_q$ e r rappresenta la classe di utenti. Dato che nel nostro sistema esiste una sola classe di utenti e non ci sono catene, si può semplificare la formula precedente riscrivendola come:

$$e_i = p_{0i} + \sum_{j=1}^{3} e_j \cdot p_{ji}$$

L'ultima formula ci permette di ottenere il seguente sistema:

$$\begin{cases} e_1 &= p_{01} + e_2 \cdot p_{21} \\ e_2 &= e_1 \cdot p_{12} \\ e_3 &= e_2 \cdot p_{23} \end{cases} \tag{1.4}$$

da cui si ricava che $e_1 = e_2 = \frac{1}{1-p_{21}}$ e $e_3 = \frac{p_{23}}{1-p_{21}}$. Inoltre,

$$\rho_1 = \frac{\lambda \cdot e_1}{\mu_1} = \frac{\lambda \cdot S_1}{1-p_{21}},\ \rho_2 = \frac{\lambda \cdot e_2}{\mu_2} = \frac{\lambda \cdot S_2}{1-p_{21}},\ \lambda_3 = \frac{p_{23}\lambda}{1-p_{21}}$$

per quanto riguarda il delay server, avendo un numero infinito di serventi, l'utilizzazione corrisponde al numero medio di utenti nel nodo. Il throughtput dei vari nodi sarà pari a $\lambda_1 = \lambda_2 = e_1 \cdot \lambda$ e $\lambda_3 = \lambda$. Il throughtput generale del sistema (aperto) è definito come il tasso con il quale i job escono dal sistema (in questo caso dal nodo 2 o 3). In una rete di code in equilibrio, il tasso di uscita è uguale al tasso di entrata.

I risultati appena ottenuti sono utili per capire qual'è il carico massimo di utenti in arrivo affinché il sistema non si saturi (nella sezione successiva vedremo come l'utilizzazione di questi nodi varia rispetto al tasso di arrivo).

Prima di calcolare il tempo medio sperimentato dagli utenti del sito web (response time, R), è necessario calcolare il numero medio di utenti per ogni nodo. Per i nodi a singolo servente il valore può essere calcolato come la somma pesata del numero di utenti per la probabilità che il nodo abbia esattamente quella cardinalità, $\sum_{k=1}^{\infty} k \cdot p_k$, oppure in alternativa come di seguito:

$$N_1 = \frac{\rho_1}{1-\rho_1} = \frac{\lambda S_1}{1-p_{21}-\lambda S_1}, \qquad N_3 = \frac{p_{23}\lambda S_3}{1-p_{21}-p_{23}\lambda S_1} \tag{1.5}$$

e rispettivamente

$$N_2 = \frac{\lambda S_2}{1-p_{21}} \tag{1.6}$$

per il delay server. Con le formule N_i siamo in grado di stabilire qual'è il carico massimo di utenti ammissibili in funzione dei tempi di servizio S_1 e S_3 e delle probabilità di diramazione p_{21} e p_{23}.

A questo punto abbiamo tutti gli elementi necessari al calcolo del response time medio che può essere calcolato come

$$R = \frac{1}{\lambda}\sum_{i=1}^{3} N_i = \frac{S_1}{1 - p_{21} - \lambda S_1} + \frac{S_2}{1 - p_{21}} + \frac{p_{23}S_3}{1 - p_{21} - p_{23}\lambda S_3} \tag{1.7}$$

e questo rappresenta proprio il tempo sperimentato dagli utenti del sito web.

Per questo tipo di sistema, definito in letteratura come transazionale, si possono calcolare alcuni limiti degli indici prestazionali. Noi calcoleremo il valore ottimistico del throughput, che indica il numero massimo di arrivi che il sistema può supportare senza saturazione, ed il valore ottimistico del response time che rappresenta appunto il tempo che un utente può sperimentare entrando nel sito. Altri indici di performance possono essere facilmente calcolati a partire dai risultati ottenuti da questi due, utilizzando le leggi fondamentali (p.e. Little's Law).

Per quanto riguarda il throughput massimo, vale che

$$X(\lambda) \leq \frac{1}{D_{max}}$$

dove max rappresenta l'indice del nodo con domanda di servizio (S_1 o S_3) più alta, per cui $max \in \{1, 3\}$ (delay server escluso) e ovviamente D_{max} è il valore stesso. Non ci resta altro che, data la precedente disequazione, calcolare il nodo che ha il demand service massimo osservando che:

$$D_1 = \frac{\rho_1}{\lambda} = \frac{S_1}{1 - p_{21}}, \qquad D_3 = \frac{\rho_3}{\lambda} = \frac{p_{23}S_3}{1 - p_{21}}$$

per cui

$$D_{max} = \max\left(\frac{S_1}{1 - p_{21}}, \frac{p_{23}S_3}{1 - p_{21}}\right)$$

e quindi

$$X(\lambda) \leq \frac{1}{max\left(\frac{S_1}{1-p_{21}}, \frac{p_{23}S_3}{1-p_{21}}\right)} \tag{1.8}$$

Per quanto riguarda invece il response time, possiamo solamente calcolare qual'è il valore ottimistico, cioè minimo, che sarà pari a $S_1 + S_2$ perché un utente può essere servito istantaneamente al nodo 1, "pensare" per S_2 secondi ed uscire dal sistema.

1.3.2 Modello Analitico con assunzione sui parametri

In questa sezione viene valutato il modello analitico presentato precedentemente facendo le seguenti assunzioni:

- Il nodo 1 ha un tempo di servizio di media S_1 pari a $1/\mu_1 = 0.02$ secondi per cui il service rate è pari a $\mu_1 = 50$.
- Il nodo 3 ha un tempo di servizio di media S_3 pari a $1/\mu_3 = 0.1$ secondi, quindi $\mu_3 = 10$.

- Think time $Z = 15$ secondi nel nodo 2.
- Probabilità di transizione $p_{21} = 0.7, p_{23} = 0.3$ per cui $p_{30} = 0$. Questo vuol dire che tutti gli utenti che entrano nel sistema per uscire devono obbligatoriamente acquistare qualche prodotto.

Il modello del sistema con le assunzioni appena elencate è rappresentato in figura 1.5.

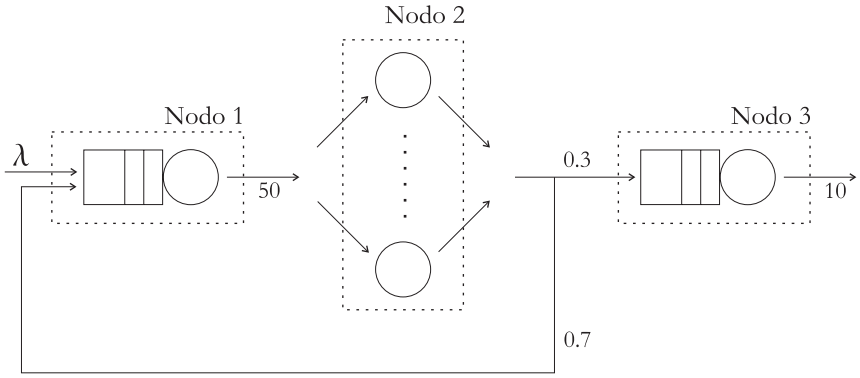


Figura 1.5: Modello a rete di code del sistema con assunzioni numeriche sui parametri S_1, S_2 e sulle probabilità di transizione.

Per ricavare le prestazioni del sistema dobbiamo "istanziare" le equazioni (1.4)-(1.7) con i valori presentati precedentemente. Per quanto riguarda il tasso di arrivo di ogni nodo, risolvendo il sistema (1.4) si ottengono i seguenti valori: $\lambda_1 = \lambda_2 = \lambda/0.3$, $\lambda_3 = \lambda$. L'utilizzazione dei vari nodi è calcolata come il rapporto tra λ_i e μ_i per cui si ottengono i seguenti risultati:

$$\rho_1 = \frac{\lambda_1}{\mu_1} = \frac{\lambda}{15}, \qquad \rho_2 = N_2 = 50\lambda, \qquad \rho_3 = \frac{\lambda}{10}$$

Come in precedenza, calcoliamo il numero medio di utenti di ogni nodo:

$$N_1 = \frac{\rho_1}{1-\rho_1} = \frac{\lambda}{15-\lambda}, \qquad N_2 = \rho_2, \qquad N_3 = \frac{\lambda}{10-\lambda}$$

e

$$N = \sum_{k=1}^{3} N_i = \frac{25(\lambda-12)}{(10-\lambda)(\lambda-15)} + 50\lambda - 2$$

Successivamente, calcoliamo il response time medio di ogni nodo che è pari a:

$$R_1 = \frac{1/\mu_1}{1-\rho_1} = \frac{3}{10(15-\lambda)}, \qquad R_2 = 50, \qquad R_3 = \frac{1}{10-\lambda} \tag{1.9}$$

ed il response time del sistema:

$$R = \frac{1}{\lambda}\left(\frac{\lambda}{15-\lambda} + \frac{\lambda}{0.02} + \frac{\lambda}{10-\lambda}\right) = \frac{50x^2 - 1252x + 7525}{(x-10)(x-15)}$$

Possiamo successivamente calcolare il response time di ogni nodo che è:

Inoltre la lunghezza media delle code che si formano è pari a:

$$Q_1 = \frac{\rho_1^2}{1-\rho_1} = \frac{\lambda^2}{15(15-\lambda)}, \qquad Q_2 = 0, \qquad Q_3 = \frac{\lambda^2}{10(10-\lambda)}$$

nonché il tempo medio di attesa:

$$W_1 = \frac{\rho_1/\mu_1}{1-\rho_1} = \frac{\lambda}{50(15-\lambda)}, \qquad W_2 = 0, \qquad W_3 = \frac{\lambda}{10(10-\lambda)}$$

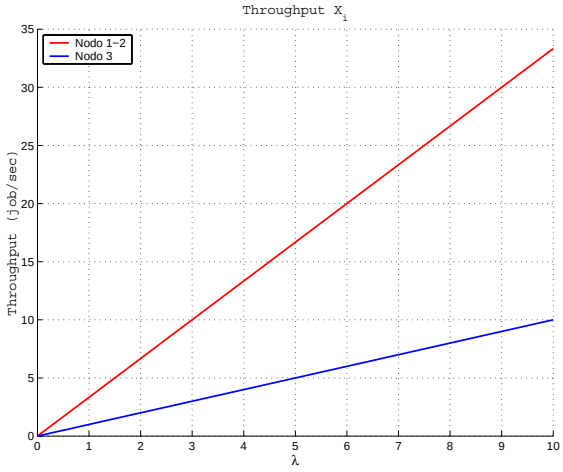
Nella figura 1.6 e 1.7 sono stati rappresentati i grafici degli indici di prestazione, tutti in funzione del tasso di arrivo λ.

1.3.3 Analisi Bound e bottleneck removal

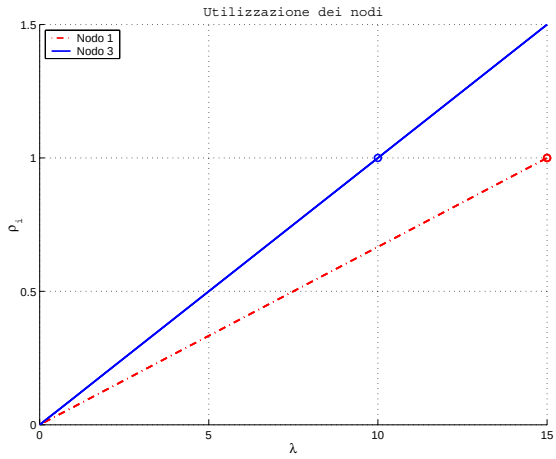
A questo punto ci chiediamo quali possono essere i vincoli che devono essere soddisfatti affinché il tempo di risposta medio del nodo 1 sia inferiore a 0.6 secondi mentre quello al nodo 3 inferiore a 2 secondi. Per quanto riguarda il primo nodo, una possibile soluzione potrebbe essere quella di vincolare il numero massimo di utenti gestibili dal primo nodo, mettendo a sistema l'equazione di R_1 (equazione 1.9) con $R_1 = 0.6$ secondi in modo tale da ottenere un carico massimo pari a 14.5 job/sec e nel caso del servente 3 il valore dovrebbe essere inferiore a 9.5 job/sec.

Da come ci si può rendere conto dal grafico dell'utilizzazione rispetto al tasso di arrivo (vedi figura 1.6(b)), il servente che risulta collo di bottiglia è il webserver sicuro perché raggiunge più velocemente, rispetto al webserver normale, il valore di saturazione.

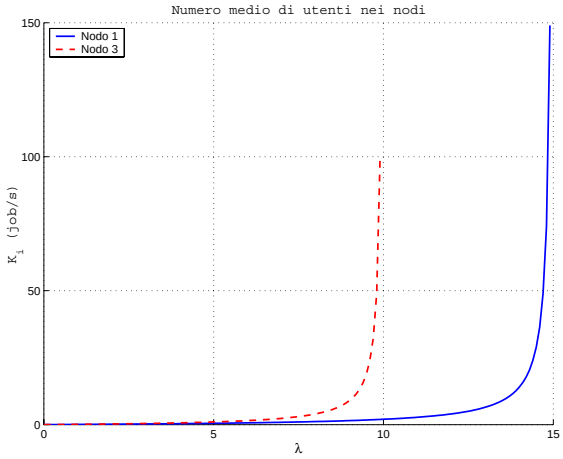
Se si raddoppia quindi la capacità del servente numero 3, in termini di dimezzamento del tempo medio di servizio S_3 (portandolo a $S_3 = 50$ msec, $\mu_3 = 20$) allora la situazione per quanto riguarda l'utilizzazione sarà quella rappresentata in figura 1.8 in cui il nuovo collo di bottiglia diventa il nodo 1 e che la velocità massima di arrivo dei job dall'esterno è pari a 15 job/sec (valore escluso).



(a)



(b)



(c)

Figura 1.6: (a) Throughput X_i. Si può vedere dal grafico che il nodo numero 3 ha un throughtput inferiore agli altri due: sarà il bottleneck del sistema. (b) Utilizzazione ρ_i: la retta che rappresenta l'utilizzazione del nodo 3 al variare del tasso di arrivo λ ha un'inclinazione maggiore rispetto a quella degli altri due nodi, indice, ancora una volta di possibile bottleneck. Il nodo 2 non è stato rappresentato perché con questa scala sarebbe stato parallelo a $x = 0$. (c) Numero medio di utenti N_i. Come si intuisce dal grafico, in tutti e due i casi, quando ci si avvicina al valore λ di saturazione del sistema, il numero medio di utenti cresce esponenzialmente.

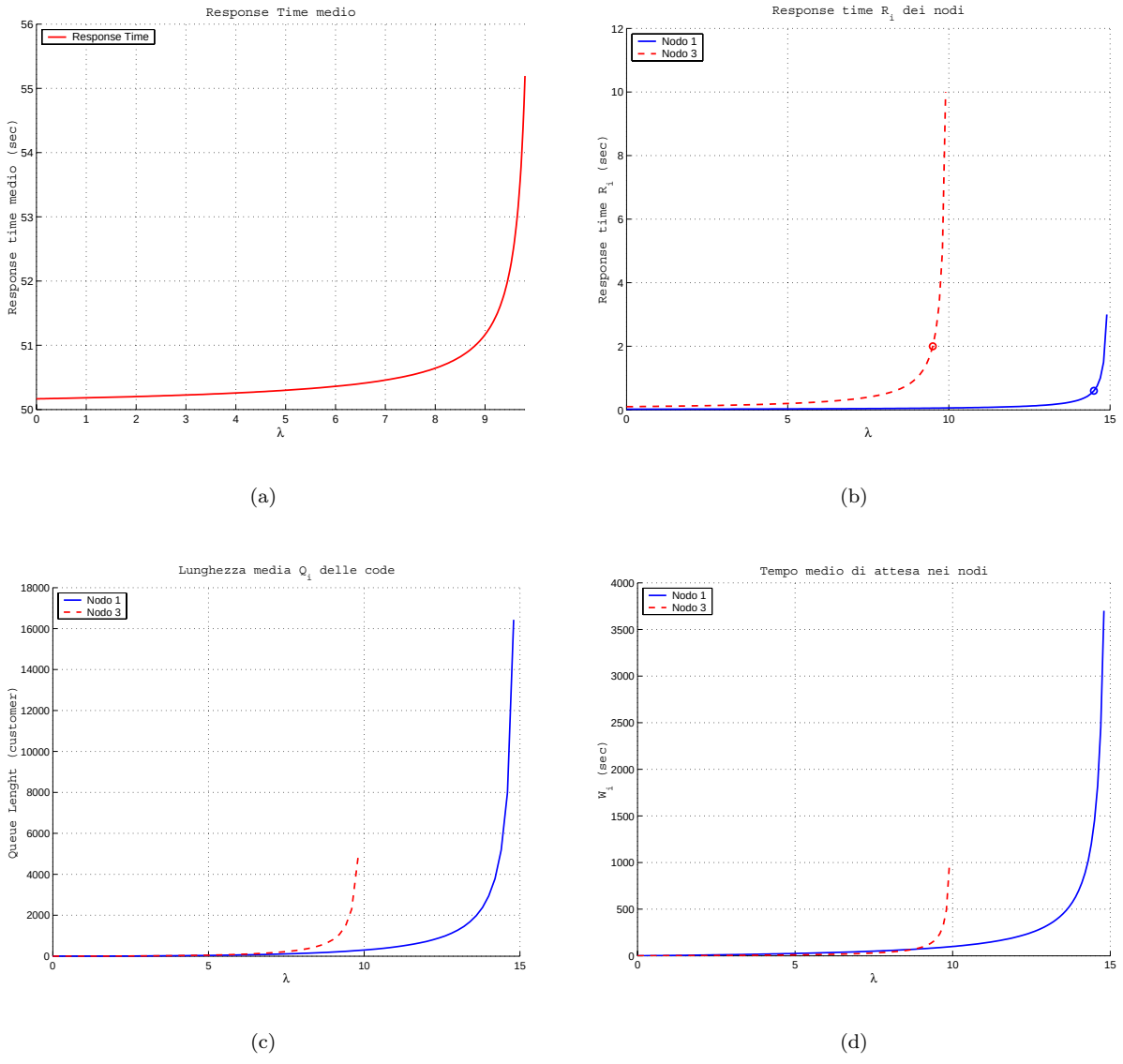


Figura 1.7: (a) Response Time medio R del sistema. Man mano che ci si avvicina al massimo numero di arrivi consentiti prima della saturazione, R aumenta esponenzialmente. (b) Response time R_i dei nodi. Il response time del nodo 2 corrisponde al think time, 15 secondi ed è costante al variare di λ. Nel grafico sono cerchiati i limiti massimi di λ se si vuole mantenere gli R_i al di sotto di una certa soglia. (c) Lunghezza media Q_i delle code nei nodi. (d) Tempo medio di attesa in coda nei nodi. L'attesa nel delay server, ovviamente, è pari a 0.

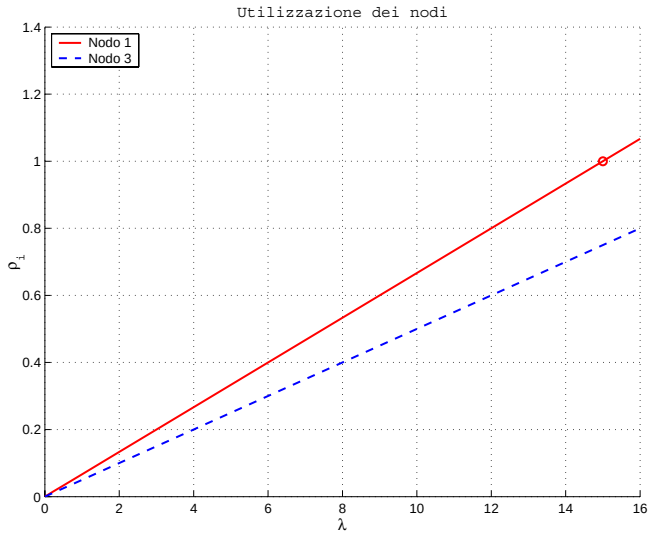


Figura 1.8: Utilizzazione dei nodi della rete di code dopo aver raddoppiato la capacità del servente al nodo 3. È stato evidenziato con un cerchio il nuovo punto di saturazione del nodo 1.

Capitolo 2

Secondo Compito

In questa seconda parte verrà presentato ed analizzato in dettaglio uno studio di simulazione (dinamica e stocastica) per il sistema presentato nella sezione 1.3. Successivamente, al sistema presentato verranno apportate alcune modifiche in termini di disciplina di servizio utilizzata, di distribuzione del tempo di servizio e di probabilità di diramazione in modo tale da analizzare come il sistema simulato si comporta con queste nuove assunzioni. Infine, verranno confrontati i risultati ottenuti rispetto ai limiti alle prestazioni ottenute nel primo compito e rispetto ad altre assunzioni stabilite. Presentiamo una ad una (sempre se possibile) le fasi dello studio di simulazione.

2.1 Formulazione del problema

Il sistema che ora prenderemo in considerazione è quello rappresentato in figura 1.5. É un sistema che astrae/idealizza un sito di commercio elettronico in cui ci sono due serventi: uno che ha il compito di visualizzare i prodotti mentre il secondo, webserver sicuro, viene usato per la procedura di acquisto. Il cliente visualizza uno o più prodotti e dopo aver pensato (think time) acquista un prodotto. In letteratura viene classificato come sistema aperto. Le assunzioni che vengono fatte per questo sistema sono le seguenti:

- Il sistema è aperto ed esiste una sola classe di utenti.
- La disciplina di scheduling dei serventi con coda che visualizzano e gestiscono la transazione economica è FIFO (First-In First-Out);
- Il think time viene implementato con un delay server i cui tempi di servizio sono esponenziali di media $S_2 = 15$ secondi.
- La distribuzione del tempo di servizio è esponenziale in tutti gli altri nodi ed ha media $S_1 = 0.02$ secondi e $S_3 = 0.1$ secondi rispettivamente per i due serventi.
- La probabilità che un utente dopo aver pensato ritorni a vedere un altro prodotto è fissata a 0.7, mentre la probabilità che un utente dopo aver visionato uno o più prodotti prosegua all'acquisto

è pari a 0.3. Questo vuol dire che si sta modellando un sistema in cui ogni cliente sicuramente acquista un prodotto.

- Il tempo di interarrivo è esponenziale e si suppone sia pari a 0.2 e 0.3. Gli utenti arrivano al sistema *uno* alla volta.
- All'inizio della simulazione non ci sono job nel sistema (sistema vuoto) e tutti i serventi sono in stato *idle*. Questa assunzione implica che si sta simulando un sito di commercio elettronico dal suo primo istante di vita.
- Il primo utente non arriva al tempo 0.
- La simulazione è del tipo *a termine*: l'evento che determina la fine è il raggiungimento di un istante di tempo fissato.
- Ogni run partirà con le stesse condizioni iniziali.

Gli obiettivi di questo studio saranno quelli di analizzare le seguenti variabili in condizioni di stabilità del sistema:

- Throughtput (X),
- Utilizzazione dei serventi (ρ),
- Tempo medio di risposta e tempo medio di attesa per richiesta ($E[T_q], E[T_w]$),
- Varianza del tempo di risposta e del tempo di attesa per richiesta ($V[T_q], V[T_w]$),
- Distribuzione del numero di richieste nel sistema in condizioni di stabilità,
- Distribuzione del numero di richieste ai singoli webserver.

Per quanto riguarda le fasi successive, astrazione del modello concettuale e raccolta dei dati, vengono tralasciate perché ampiamente trattate nella sezione 1.3 e successive.

2.2 Codifica del Modello

Questa parte dello studio di simulazione consiste nella trasformazione del modello concettuale in una forma interpretabile dal calcolatore ottenendo quindi un modello operazionale.

Esistono fondamentalmente quattro metodologie di simulazione possibili: orientata ai processi, scheduling di eventi, scansione di attività e metodo a tre fasi. La nostra scelta ricade sul primo. In questa tipologia di simulatori, *il sistema viene descritto in termini di processi che sono eseguiti in parallelo e che interagiscono tra di loro scambiandosi informazioni*. Questa definizione ci obbliga ad identificare quali sono i processi che entrano in gioco. Nel nostro caso, nel sistema si possono identificare i seguenti processi:

- Il Controller (detto anche Scheduler),

- Il servente per la visualizzazione dei prodotti, View Product Server (in seguito abbreviato VPServer o VPS),
- Il servente utilizzato per l'acquisto sicuro dei prodotti, Buy Product Server (abbreviato BpServer o BPS),
- Il Delay Server,
- Il processo Sampler per il campionamento delle variabili da osservare,
- Il processo RestartCounters,
- Il processo Arrivals.

Il Controller è quel processo che gestisce l'esecuzione di tutti i processi, mantenendo una coda dei processi che devono essere attivati e lanciandoli rispettando l'ordine di partenza prevista. I processi VpServer, Delay server ed BpServer rappresentano i rispettivi server come nel modello analitico; il processo Sampler si occupa, ad intervalli fissi di campionare le variabili da osservare; il processo RestartCounters viene invocato una volta sola alla fine della fase transiente e si occupa, come dice il nome stesso, di azzerare tutti i contatori in modo tale che le oscillazioni di questa fase non influenzino i risultati finali. Infine, il processo Arrivals modella gli utenti che arrivano al sito.

Tutti i processi appena presentati possono trovarsi in uno dei seguenti stati:

- Attivo se è attualmente in esecuzione (non si trova nella coda dello scheduler),
- Sospeso se è presente nella coda dello scheduler in attesa che venga raggiunto l'istante di attivazione.
- Passivo se il processo non è presente nella coda di schedulazione e rimarrà tale fin tanto che un altro processo non lo attiverà.
- Terminato se il processo non si trova nella coda di schedulazione e non potrà mai esserlo in futuro.

Per quanto riguarda l'implementazione del simulatore si è scelto di utilizzare la libreria JavaSim, libreria completa e largamente utilizzata sviluppata all'università di NewCastle upon Tyne, UK. Questa libreria è l'implementazione Java del toolkit C++Sim utilizzato per la simulazione discreta orientata agli oggetti. Questa libreria è stata consigliata dall'esperienza maturata da altri studenti che hanno avuto la necessità di utilizzare questo strumento. Non rientra nello scopo di questo lavoro la presentazione di tutte le caratteristiche della libreria, nel qual caso si rimanda la consultazione del sito ufficiale [8].

Per quanto riguarda il nostro sistema, tutti i processi definiti come tali in precedenza (che sono rappresentati rispettivamente da classi java) devono estendere la classe SimulationProcess di JavaSim in modo tale da ereditarne le caratteristiche.

2.2.1 Struttura del simulatore

Elenchiamo quali sono le principali azioni che effettuano le componenti del simulatore che abbiamo creato.

Processo Arrivals: Inizialmente genera un job e lo mette in coda al webserver 1. Verranno aggiornati i contatori, come per esempio il numero di job arrivati dall'esterno, il numero di job che sono presenti nel sistema, il numero di job presenti al nodo 1. Verrà inoltre creato il valore dell'interarrivo del job successivo rispetto alla distribuzione di riferimento.

Processo VpServer: Se la coda del nodo ha qualche elemento, estrae il primo (rispetto alla politica di scheduling definita) e viene generato il tempo di servizio necessario per il job corrente. Verranno aggiornati i contatori che gestiscono la media e la varianza dei tempi di servizio.

Processo DelayServer: Quando un job esce dal nodo che sta a monte, viene generato il tempo di servizio richiesto, vengono aggiornati i contatori per le statistiche e, dopo che il job ha terminato l'elaborazione, in base alla probabilità di diramazione sarà trasferito (cioè inserito nelle rispettive code) al nodo di partenza (VPS) oppure al nodo successivo (BPS) rispettivamente per la visione di un altro prodotto o per l'acquisto di un prodotto.

Processo BpServer: Le azioni da compiere sono circa identiche a quelle del processo VpServer (con le dovute modifiche alla distribuzione del tempo di servizio). Alla fine dell'elaborazione, i job escono dal sistema e le statistiche vengono aggiornate di conseguenza (per esempio il numero di utenti del sistema, il response time, ...)

Processo Sampler: Questo processo viene invocato ad intervalli prefissati (ogni 10 secondi) in modo tale da registrare su file il valore corrente del numero di utenti nel sistema. Questo processo viene utilizzato solamente nella fase di studio della lunghezza della fase transiente (test di Welch).

Oltre a tutte le classi che rappresentano i processi, nel progetto sono state utilizzate delle classi aggiuntive necessarie al completo svolgimento del progetto. Ne elenchiamo brevemente nomi e caratteristiche.

- Queue, utilizzata per la gestione delle code nel VpServer e BpServer. Al suo interno, ha implementato un algoritmo che estrae gli elementi in modo casuale o FIFO.
- ArrayDS, classe che implementa il concetto di array (che nel nostro caso sarà utilizzata come array di Delay Server). É stato deciso di implementare il delay server usando un array di processi in modo tale che se un job ha bisogno di un servente, se ne ricerca uno di libero nell'array prima di crearne uno di nuovo. Questa ovviamente non è l'unica tecnica disponibile, ma è l'unica che, dalle nostre prove, fa un uso limitato dei thread.
- Job, classe che implementa il concetto di cliente. Nella classe sono stati definiti diversi campi che sono necessari ad identificare le caratteristiche del job: numero di visite effettuate al VPS, tempo di arrivo in coda...),
- Main, classe che fa partire il simulatore e che gestisce l'import di parametri come per esempio i semi che vengono utilizzati nel run.

2.3 Verifica e validazione del modello

Con *verifica* di un simulatore si intende quel processo che determina se le assunzioni del sistema sono state correttamente trasformate in un programma ed è stata effettuata facendo il debug.

La *validazione*, nella fase successiva, si basa sulla seguente tecnica [7]: si esegue il simulatore con varie impostazioni dei parametri di input e si analizza l'output se questo è fattibile. Per tasso di arrivo pari a 0.12 i risultati ottenuti sono quelli presentati nella tabella 2.2. Da come si può confrontare con la tabella dei risultati analitici (vedi tabella 2.1), la validazione si considera superata con successo.

μ	$\lambda = \lambda_3$	λ_1, λ_2	ρ_1	ρ_2	ρ_3	N_s	$E[T_q]_1$	$E[T_q]_3$	$E[T_w]_1$	$E[T_w]_3$	$E[Q]_1$	$E[Q]_3$
0.11	9.0909	30.3030	0.6061	454.5455	0.9091	466.0839	0.0507692	1.100000	0.0307692	1.0000	0.9324	9.0909
0.13	7.6923	25.6410	0.5125	384.6154	0.7695	389.0013	0.0410526	0.433333	0.0210526	0.3333	0.5398	2.5641
0.15	6.6667	22.2222	0.4444	333.3333	0.6667	336.1333	0.0360000	0.300000	0.0160000	0.2000	0.3556	1.3333
0.17	5.8824	19.6078	0.3922	294.1176	0.5882	296.1914	0.0329032	0.242857	0.0129032	0.1429	0.2530	0.8403
0.19	5.2632	17.5439	0.3509	263.1579	0.5263	264.8095	0.0308108	0.211111	0.0108108	0.1111	0.1897	0.5848
0.20	5.0000	16.6667	0.3333	250.0000	0.5000	251.5000	0.0300000	0.200000	0.0100000	0.1000	0.1667	0.5000
0.21	4.7619	15.8730	0.3175	238.0952	0.4762	239.4694	0.0293023	0.190909	0.0093023	0.0909	0.1477	0.4329
0.22	4.5455	15.1515	0.3030	227.2727	0.4545	228.5408	0.0286957	0.183333	0.0086957	0.0833	0.1318	0.3788
0.23	4.3478	14.4928	0.2899	217.3913	0.4348	218.5687	0.0281633	0.176923	0.0081633	0.0769	0.1183	0.3344
0.24	4.1667	13.8889	0.2778	208.3333	0.4167	209.4322	0.0276923	0.171429	0.0076923	0.0714	0.1068	0.2976
0.25	4.0000	13.3333	0.2667	200.0000	0.4000	201.0303	0.0272727	0.166667	0.0072727	0.0667	0.0970	0.2667
0.26	3.8462	12.8205	0.2564	192.3077	0.3846	193.2775	0.0268966	0.162500	0.0068966	0.0625	0.0884	0.2404
0.27	3.7037	12.3457	0.2469	185.1852	0.3704	186.1013	0.0265574	0.158824	0.0065574	0.0588	0.0810	0.2179
0.28	3.5714	11.9048	0.2381	178.5714	0.3571	179.4395	0.0262500	0.155556	0.0062500	0.0556	0.0744	0.1984
0.29	3.4483	11.4943	0.2299	172.4138	0.3448	173.2386	0.0259701	0.152632	0.0059701	0.0526	0.0686	0.1815
0.30	3.3333	11.1111	0.2222	166.6667	0.3333	167.4524	0.0257143	0.150000	0.0057143	0.0500	0.0635	0.1667

Tabella 2.1: Valori teorici delle variabili del modello analitico studiato

	Sistema		
X	8.30923333		
N	421.90929188		
	VPS	**DS**	**BPS**
X	27.71143333	27.71143333	8.30923333
ρ	0.55438668	416.99292383	0.83226271
E[T_s]	0.0200057	14.97218984	0.10016119
V[T_s]	3.99969114e-4	224.02974115	0.01005744
E[T_w]	0.02487341	0.0	0.49807511
V[T_w]	0.00161825	0.0	0.33634094
E[T_q]	0.04487911	14.97218984	0.59823783
V[T_q]	0.00202126	224.02974115	0.34613732

Tabella 2.2: Tabella per la validazione del simulatore. E e V indicano rispettivamente media e varianza, T_s tempo di servizio, T_w tempo di attesa e T_q tempo di risposta.

2.4 Progetto degli esperimenti di simulazione

Il progetto degli esperimenti di simulazione consiste nel definire:

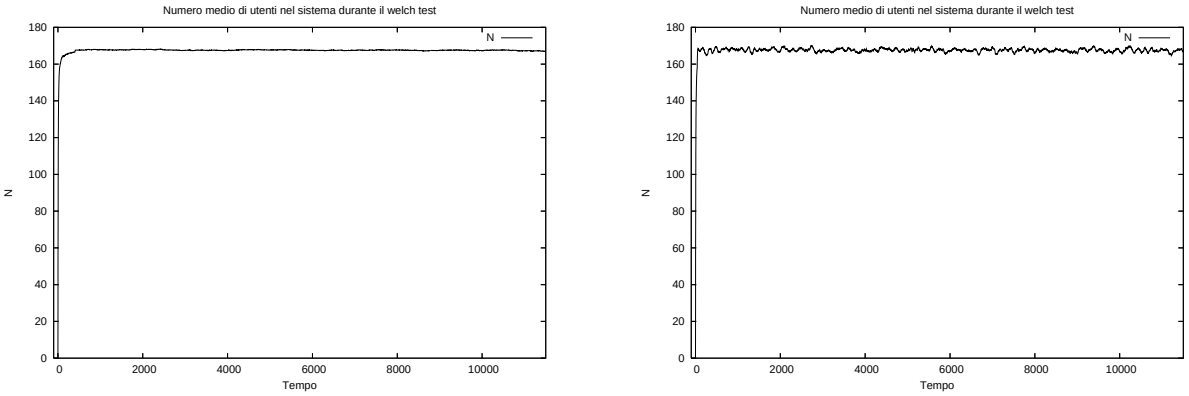
- La lunghezza di ogni run,
- La lunghezza del periodo di warm-up,
- Il numero di run indipendenti da effettuare utilizzando di volta in volta semi diversi per i generatori di numeri casuali.

La lunghezza di ogni run è stata stabilita pari a 42 ore circa. Questa durata rappresenta un periodo molto esteso rispetto alla durata della fase transiente. Per quanto riguarda la lunghezza della fase transiente, si è utilizzato il metodo proposto da Welch, il quale si basa nell'effettuare n esperimenti (run) e per ognuno di questi campionare m volte una determinata variabile (solitamente quella più instabile). L'esecuzione del Welch test consente di ottenere una matrice Y di dimensione $n \times m$ in cui l'elemento Y_{ji} rappresenta l'osservazione i-esima della replicazione j-esima. Successivamente, vanno calcolate le medie $\overline{Y}_i = \sum_{j=1}^{n} Y_{ji}/n$ per i=1,2,...,m. Infine, per livellare le oscillazioni di alta frequenza presenti in $\overline{Y}_i$ viene definita la media mobile $\overline{Y}_i(w)$ come segue:

$$Y_i(w) = \begin{cases} \frac{\sum_{s=-w}^{w} \overline{Y}_{i+s}}{2w+1} & \text{se } i = w+1, ..., m-w \\ \frac{\sum_{s=-(i-1)}^{i-1} \overline{Y}_{i+s}}{2i-1} & \text{se } i = 1, ..., w \end{cases}$$

Alla fine plottando gli Y_i al variare del tempo, si identifica un valore di l superato il quale la variabile in questione, graficamente parlando, assume un comportamento stabile (circa costante). Abbiamo stabilito di effettuare 20 run (run pilota, diversi dai run effettuati per l'analisi del comportamento del simulatore)

ed in ognuno campionare 14.500 osservazioni della variabile N, "numero di utenti nel sistema". La finestra w utilizzata per il livellamento è stata impostata a 400. I risultati ottenuti sono rappresentati in figura 2.1, per cui si è stimato che la durata della fase transiente è di 10.000 unità di tempo.



(a) (b)

Figura 2.1: Test di Welch. Andamento del numero di utenti del sistema con i parametri n=20, m=14.500. (a) w=400, (b) w=40. Si può notare benissimo come una diminuzione della finestra porti inevitabilmente ad un andamento della curva meno costante.

Per quanto riguarda il numero di run da effettuare, si è stabilito un valore pari a 100. Inoltre, ad ogni run vengono utilizzati semi diversi generati dal simulatore alla fine del run precedente.

2.5 Esecuzione ed analisi dei dati

Per l'analisi dei dati è stato utilizzato il metodo delle prove ripetute. Questo consiste nel ripetere l'esperimento n volte ($n = 100$), eliminando di volta in volta le informazioni relative alla fase transiente. Inoltre ad ogni prova i vengono raccolti k_i osservazioni in modo tale che ogni k_i sia diverso uno dall'altro. Successivamente, si effettua la media dei valori osservati per ogni run (medie per colonne) e si considerano questi valori ottenuti come elementi del campione. Questo campione, a differenza di quelli ottenuti nello stesso run, sono i.i.d. e quindi si può applicare l'analisi statistica classica per la stima, per esempio, della media e della varianza di un indice.

Dopo aver eseguito il simulatore e raccolto tutti i dati prodotti, bisogna convalidare i dati ottenuti. La *convalida* è necessaria per stabilire se il modello nel suo dominio di applicabilità è sufficientemente accurato per l'applicazione prevista. Ci sono diversi modi per effettuare la convalida, che dipendono essenzialmente da quale sistema/modello usiamo per fare il confronto. Per esempio, si può utilizzare un sistema reale, un modello matematico o un altro simulatore. Nel nostro caso, avendo a disposizione solamente un modello analitico del sistema, la convalida viene fatta rispetto a questo. La convalida consiste nell'eseguire il simulatore sotto le stesse ipotesi del modello analitico, estrarre un campione da

questo, effettuare una stima della grandezza in esame, generando un intervallo di confidenza: se il valore teorico che deriva dal modello analitico cade all'interno dell'intervallo di confidenza allora il confronto da esito positivo.

Prima di tutto vediamo come sono state ottenute le stime puntuali e gli intervalli di confidenza. Grazie al teorema del limite centrale possiamo affermare che, per grandi valori di n (come nel nostro caso), la statistica

$$Z = \frac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}}$$

ha approssimativamente distribuzione normale standardizzata. Rispetto alla notazione usata in precedenza, μ indica la media della popolazione. Se l'area sotto la curva in questa distribuzione normale a destra di $z_{\frac{\alpha}{2}}$ vale $\alpha/2$, allora l'area compresa fra $-z_{\frac{\alpha}{2}}$ e $z_{\frac{\alpha}{2}}$ vale $1-\alpha$, quindi

$$P(-z_{\frac{\alpha}{2}} < Z < z_{\frac{\alpha}{2}}) = 1 - \alpha$$

Conseguentemente, con probabilità $1-\alpha$, si può asserire che è soddisfatta la disuguaglianza:

$$-z_{\frac{\alpha}{2}} < \frac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}} < z_{\frac{\alpha}{2}}.$$

Da questa disuguaglianza, possiamo risolvere rispetto a μ ed ottenere:

$$\overline{X} - z_{\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}} < \mu < \overline{X} + z_{\frac{\alpha}{2}} \cdot \frac{\sigma}{\sqrt{n}}$$

Una volta estratto un campione di ampiezza n ($n > 30$) e calcolato il valore $\overline{x}$ della media del campione, aver calcolato lo scarto quadratico medio campionario s, si ottiene la seguente stima per intervallo per la media μ, soddisfatta con probabilità $1-\alpha$.

$$\overline{x} - z_{\frac{\alpha}{2}} \cdot \frac{s}{\sqrt{n}} < \mu < \overline{x} + z_{\frac{\alpha}{2}} \cdot \frac{s}{\sqrt{n}}. \tag{2.1}$$

In generale quindi un intervallo di confidenza con grado di fiducia del 95% va interpretato nel seguente modo: se si considerano tutti i possibili campioni di dimensione n, e per ciascuno di essi si calcola la media campionaria ed il corrispondente intervallo di confidenza centrato su questa, il 95% degli intervalli così ottenuti contiene il corrispondente parametro della popolazione e solo il 5% non lo contiene. Per quanto detto, non possiamo sapere se uno specifico intervallo contiene o meno il parametro della popolazione, tuttavia possiamo affermare che abbiamo un grado di fiducia ad esempio del 95% di aver scelto un campione a cui corrisponde una stima per intervallo comprendente il parametro della popolazione. Il campione che consideriamo in questo progetto è formato da 100 elementi (uno per ogni run), quindi un grande campione, e considereremo due valori del grado di fiducia 95% e 99% da cui derivano i seguenti valori per $z_{\frac{\alpha}{2}}$: 1.96 e 2.576.

I risultati sono stati ottenuti fissando il tempo di interarrivo medio tra un cliente ed il successivo a 0.3 (vedi tabella 2.3). Da come si può osservare, per quanto riguarda il grado di fiducia del 99%, la fase di convalida è quasi sempre andata a buon termine. Solamente in un caso il valore teorico non cade all'interno dell'intervallo di confidenza (utilizzazione del delay server), anche se è un valore comunque molto vicino all'intervallo.

Indice	Val. mod.	Conf=95%		Accetto?	Conf=99%		Accetto?
	analitico	min	max		min	max	
X	3.333333	3.3325520	3.3344776	si	3.3322495	3.3347802	si
ρ_1	0.222222	0.2221347	0.2223369	si	0.2221029	0.2223687	si
ρ_2	166.6667	167.09346	167.23317	no	167.07151	167.25513	no
ρ_3	0.333333	0.3332019	0.3335019	si	0.3331548	0.3335490	si
$E[t_q]_1$	0.025714	0.0257116	0.0257258	si	0.0257094	0.0257170	si
$E[t_q]_2$	15.00000	14.997157	15.001543	si	14.996468	15.002233	si
$E[t_q]_3$	0.150000	0.1499966	0.1501889	si	0.1499664	0.1502191	si
$E[t_w]_1$	0.005714	0.0057128	0.0057228	si	0.0057112	0.0057243	si
$E[t_w]_3$	0.050000	0.0500176	0.0501677	no	0.0499941	0.0501913	si

Tabella 2.3: Risultati della convalida dei dati

2.6 Modifica del sistema

Fin'ora tutti i risultati ottenuti riguardavano il sistema definito nel primo compito. É il caso di vedere come il sistema si comporta apportando le seguenti modifiche:

- La disciplina di scheduling del webserver principale (VpServer) è Processor Sharing (assumiamo che non sia preemptive),
- La disciplina di scheduling del webserver sicuro (BpServer) è di tipo Random,
- La distribuzione del tempo di servizio del VPS è normale con media S_1 e varianza pari al 20% di S_1.
- La distribuzione del tempo di servizio del BPS è iperesponenziale con due fasi e parametro $p = 0.6$,
- La distribuzione del tempo di servizio del DS è esponenziale,
- La probabilità di diramazione non è più costante, bensì è funzione del numero di visite già effettuate (vedi figura 2.2) e con i seguenti parametri: A=0.5, B=0.9, K1=15 e K2=50.
- Il sistema così modificato utilizza una distribuzione del tempo di servizio normale nel webserver principale. Come sappiamo, la distribuzione normale ha una forma a campana, centrata sul valore medio, cioè 20 millisecondi. La varianza è pari al 20% della media, cioè 4 millisecondi. Più il valore di varianza è alto, più la campana è bassa e larga e questo si traduce nella possibilità che il generatore di numeri casuali produca valori negativi. Si è osservato empiricamente che in alcuni casi questo può succedere (anche se la probabilità è bassa), per cui si è deciso che la soluzione migliore a questo problema, dato che tempi di servizio negativi non hanno significato, sia quella di scartarli.
- Dato che il nodo 3 ha distribuzione iperesponenziale, è necessario impostare i parametri μ_1 e μ_2. Si è utilizzato quindi le formule (1.25) e (1.26) di [6] con probabilità di diramazione $q_1 = 0.6$ e $q_2 = 0.4$

calcolando il coefficiente di variazione con i dati del progetto precedentemente convalidato (vedi appendice A).

- Tutte le assunzioni che sono state fatte in precedenza, rimangono ancora valide.

La più grossa modifica che è stata apportata a questo progetto è la gestione della disciplina del webserver principale, gestendola come processor sharing. Questa disciplina corrisponde ad un round robin in cui il quanto di tempo tende a zero. Elenchiamo brevemente come questa nuova funzionalità viene gestita dal simulatore.

Arrivo di un nuovo utente dall'esterno:

1. Genero il tempo totale di servizio richiesto t per il job corrente rispetto ad una certa distribuzione. Associo questo valore al job.
2. Imposto il valore del campo "elaborazione rimanente" (r) del job a t.

Nodo libero (un job è appena uscito):

1. Se ci sono k utenti in coda, calcolo la velocità con cui devono essere serviti i job come $q = 1/(k * \mu_1)$.
2. Assegno al primo cliente in coda il quanto q di esecuzione appena calcolato.
3. Prendo il primo cliente in coda: se l'elaborazione rimanente per questo cliente è superiore a q allora Hold(q) altrimenti Hold(r).

Al termine di un'elaborazione:

1. Aggiorno l'elaborazione rimanente del job a $r = r - q$.
2. Se $r > 0$, cioè se il job corrente non ha completato il servizio in questo nodo, accoda nuovamente il job, altrimenti manda il job al Delay Server.
3. Richiama la procedura "Nodo Libero".

Per quanto riguarda il progetto degli esperimenti di simulazione, si utilizzeranno gli stessi parametri definiti in precedenza e che riassumiamo brevemente:

- Lunghezza della fase transiente pari a 4000 unità di tempo.
- Numero di run da effettuare pari a 100.
- Durata max del simulatore pari a 48 ore.

Vediamo ora quali sono stati i risultati ottenuti, aiutandoci con i grafici 2.3, 2.4 e 2.5. Tutti i test che sono stati fatti, sono stati fatti utilizzando un tempo di interarrivo α dei job pari a 0.3 e 0.2 in modo tale che si potesse vedere come il sistema si comporta con due carichi di lavoro diversi. Nella figura 2.3 (a) per esempio si vede come si generi un throughput più alto quando il tasso di interarrivo è più basso (cioè la velocità con cui entrano gli utenti nel sistema è più alta). Questo è vero perché non si è raggiunta la

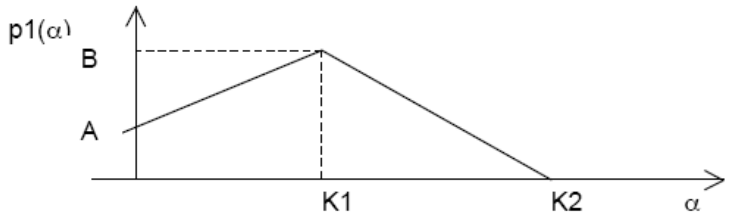


Figura 2.2: Probabilità diramazione dopo che l'utente ha pensato. Si può notare come la probabilità di visionare un nuovo prodotto è crescente inizialmente fino a quando si raggiunge il valore massimo K1 superato il quale decresce fino ad essere pari a zero (cioè quando l'utente acquisterà uno o più prodotti). I valori dei parametri A, B, K1 e K2 sono pari a $0.5, 0.9, 15, 50$ rispettivamente.

soglia di saturazione (che è pari a 0.1). Per quanto riguarda l'utilizzazione il discorso rimane lo stesso (con i rispettivi valori degli indici). Stesso discorso per l'utilizzazione, più α è basso più ρ aumenta. Nel caso del delay server, questo valore è rappresentato dal numero di utenti nel nodo.

Per quanto riguarda il tempo di risposta, si sono rilevati comportamenti normali: più job arrivano al sistema, più il tempo di risposta aumenta. Per quanto riguarda invece il delay server, dato che non c'e' coda, il valore rimane, a meno di una certa variabilità, lo stesso.

La figura 2.5 (b) rappresenta il grafico della distribuzione del numero delle richieste al sistema in condizioni di stabilità. Come si può vedere, il grafico ha una curva a campana, tipica della distribuzione normale. Nel webserver principale e in quello sicuro, la distribuzione degli utenti (numero di richieste) risulta diversa rispetto a quella del sistema. In tutti e due i casi vediamo che nella maggior parte delle volte, il numero di utenti non è superiore a qualche unità, valore corretto se andiamo a vedere qual'è il valore del throughput.

2.7 Confronto con limiti alle prestazioni

In questa sezione vengono analizzati quali sono le differenze riscontrate tra l'analisi dei limiti alle prestazioni del primo compito e i risultati ottenuti del progetto convalidato (non sarebbe possibile farlo per il progetto modificato perché il sistema è diverso). Per quanto riguarda i limiti alle prestazioni calcolati, throughput e Response Time, tutte e due le disequazioni sono rispettate. Infatti, il throughput massimo secondo la formula 1.8 è 10, per cui dai dati ottenuti questo valore è ampiamente rispettato. Per quanto riguarda invece il response time, essendo pari a 50, anche questo è ampiamente rispettato.

2.8 Confronto con probabilità costante

In questa sezione viene presentato un confronto dei risultati ottenuti col modello a probabilità di diramazione costante (0.7-0.3) rispetto al modello con probabilità dipendente dal numero di visite effettuate.

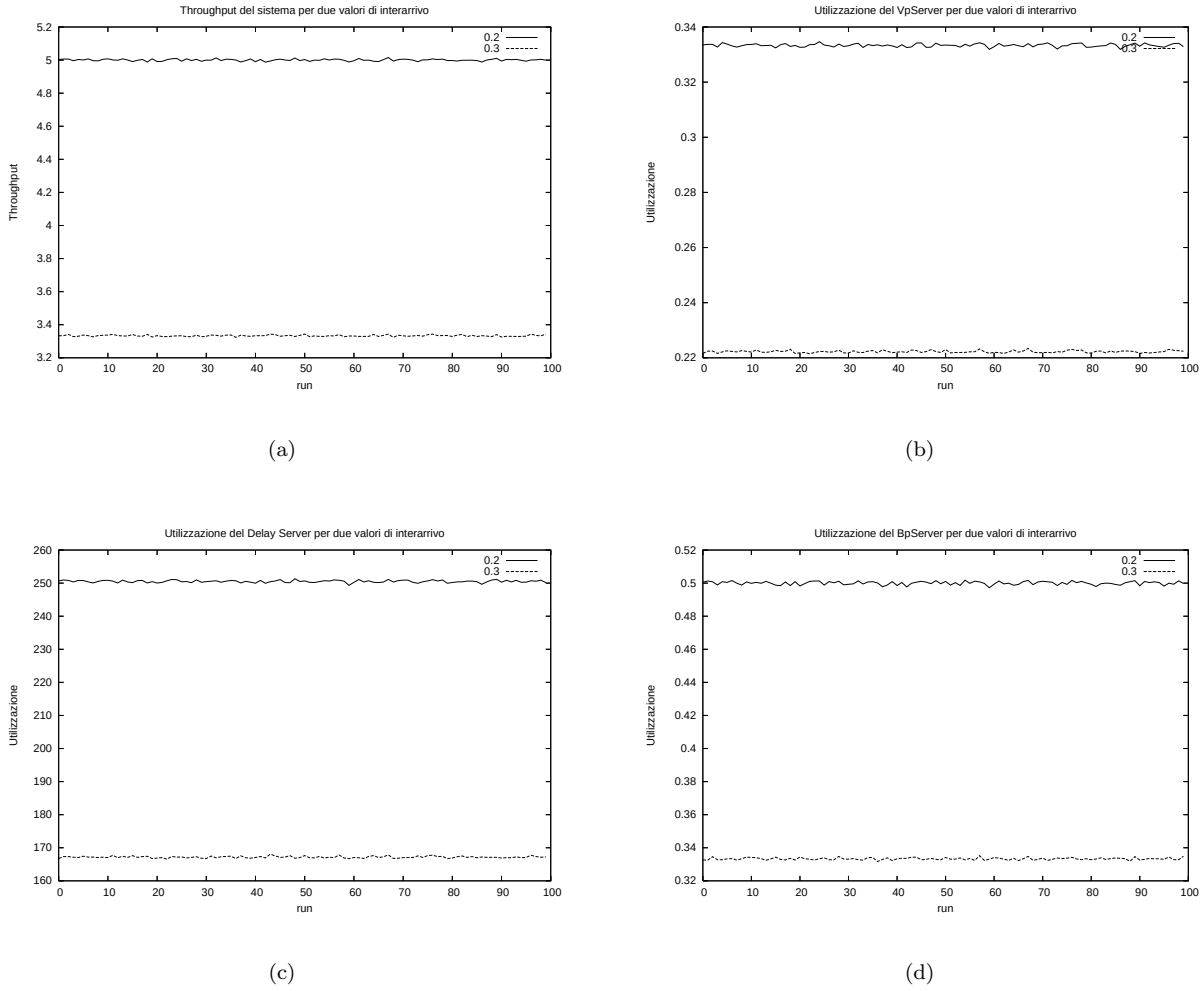
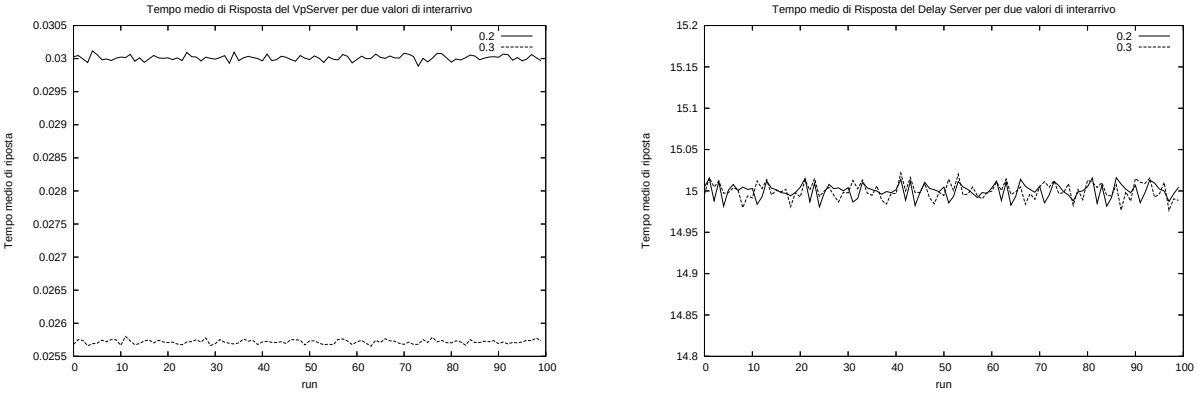
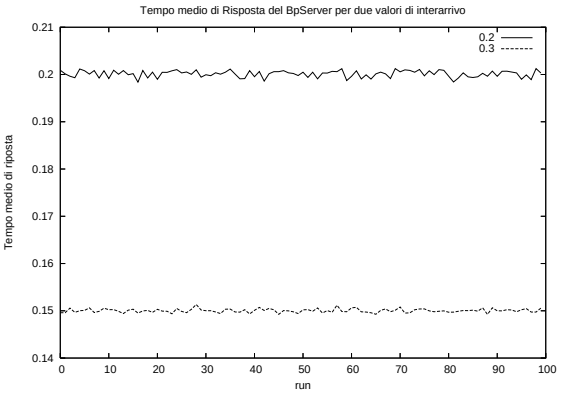


Figura 2.3: (a) Throughtput del sistema in condizioni di stabilità. Due sono i casi contemplati: il primo con tempo di interarrivo α pari a 0.3 ed il secondo a 0.2. (b),(c) e (d) Utilizzazione nei tre serventi.

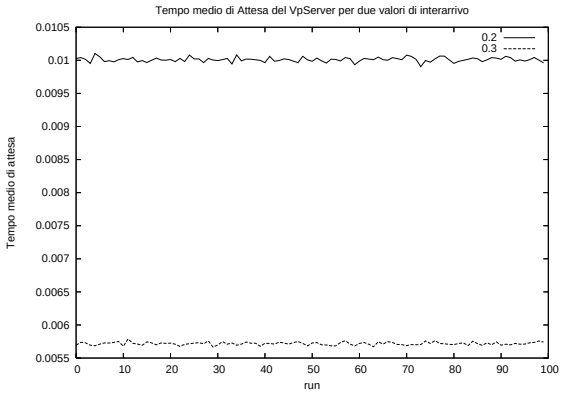


(a)

(b)



(c)



(d)

Figura 2.4: (a),(b) e (c) Tempo medio di risposta di ogni utente nei tre nodi del sistema. (d) Tempo medio di attesa al nodo 1.

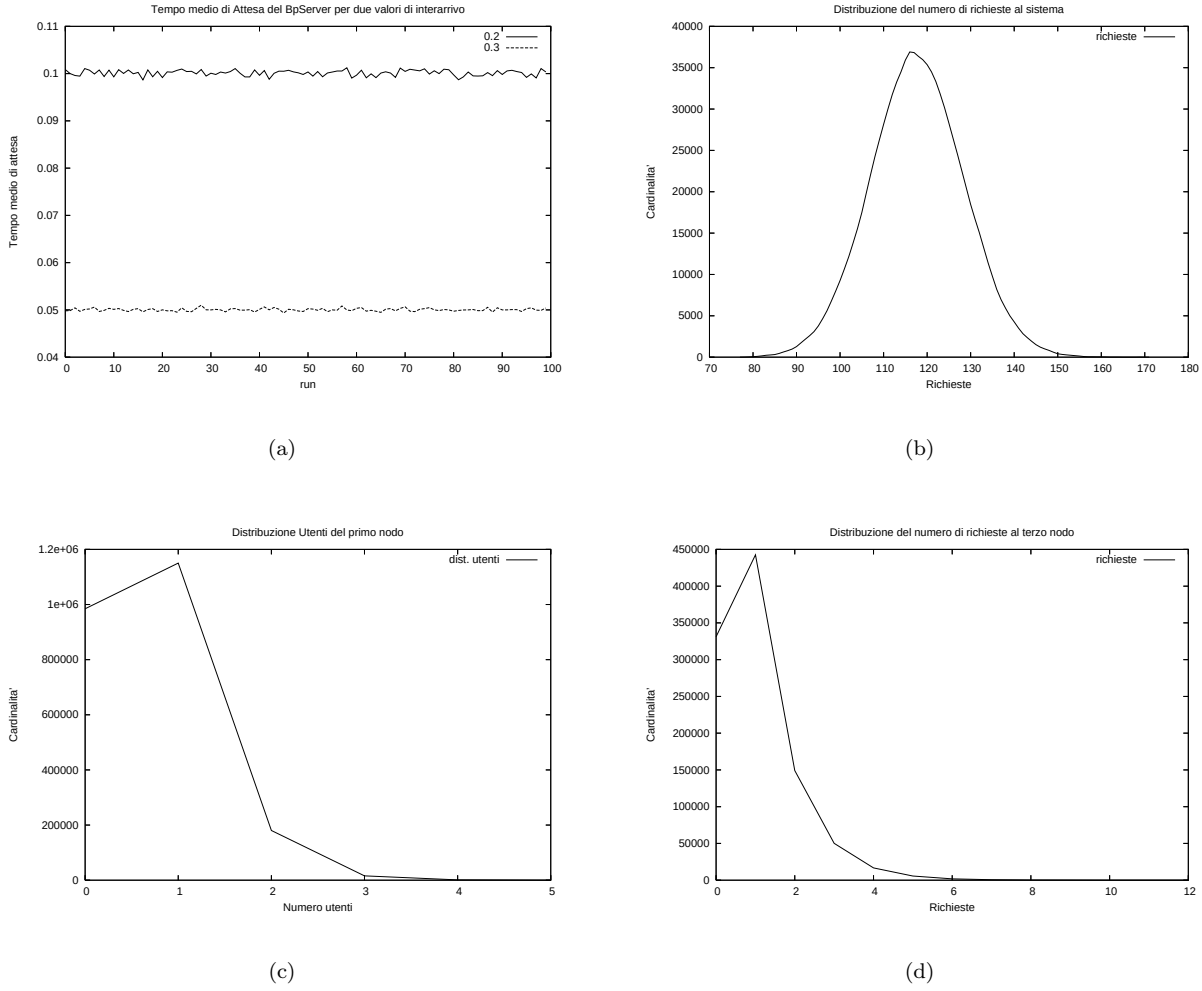


Figura 2.5: (a) Tempo medio di attesa nel nodo 3. Il nodo 2 non ha tempi di attesa dato che non ha coda e che ad ogni job in arrivo ha sempre un servente disponibile. (b) Grafico della distribuzione dei nodi quanto il sistema è stabile. Si riconosce una forma tipica della distribuzione normale. (c) e (d) Distribuzione delle richieste ai nodi 1 e 3. Dai due grafici notiamo come in (d) sia molto più probabile che un utente appena arrivato al BpServer debba attendere in coda prima di essere servito.

Vediamo ora in dettaglio quali sono le differenze riscontrate.

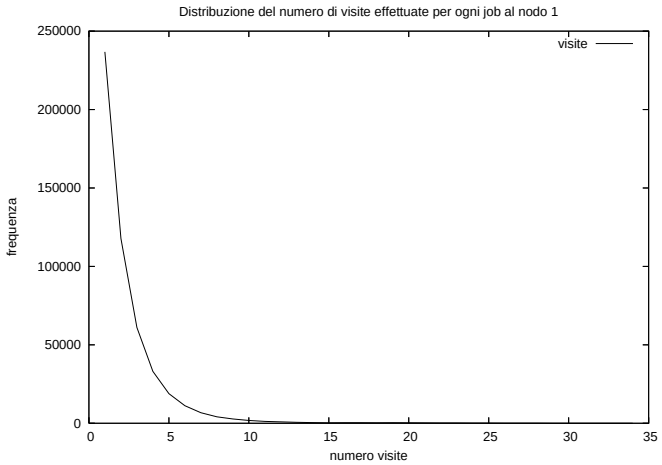


Figura 2.6: Distribuzione del numero di visite effettuate per job al webserver principale.

- Il Throughput è inalterato. Nei sistemi stabili non saturi corrisponde al tasso di arrivo per cui non ha motivo di essere diverso dalla soluzione precedente.
- L'utilizzazione è diminuita nel primo e nel secondo nodo. La nuova funzione di diramazione si comporta in maniera differente rispetto a quella costante. Dall'osservazione della figura 2.6 che rappresenta la distribuzione del numero di visite effettuate per job al webserver principale vediamo che la maggior parte dei job effettua un numero di visite abbastanza basso. Inoltre, dallo studio della funzione di figura 2.2 si può ricavare che per valori di x inferiori a 7,5 la probabilità che un cliente ritorni al primo nodo è sempre inferiore a 0.70 (stesso valore che avevamo per la funzione costante). Grazie a questo, unito alla distribuzione delle visite ricavate, se ne deduce che il numero di utenti del webserver principale sarà inferiore.
- Per il discorso fatto al punto precedente, possiamo affermare che anche il tempo di risposta ed il tempo di attesa del webserver principale è inferiore con questa nuova funzione di diramazione (ci saranno meno job in coda).
- Vedendo la figura 2.9 (b) ci si rende conto come la distribuzione del numero di richieste si sia spostata verso un valore mediano inferiore.

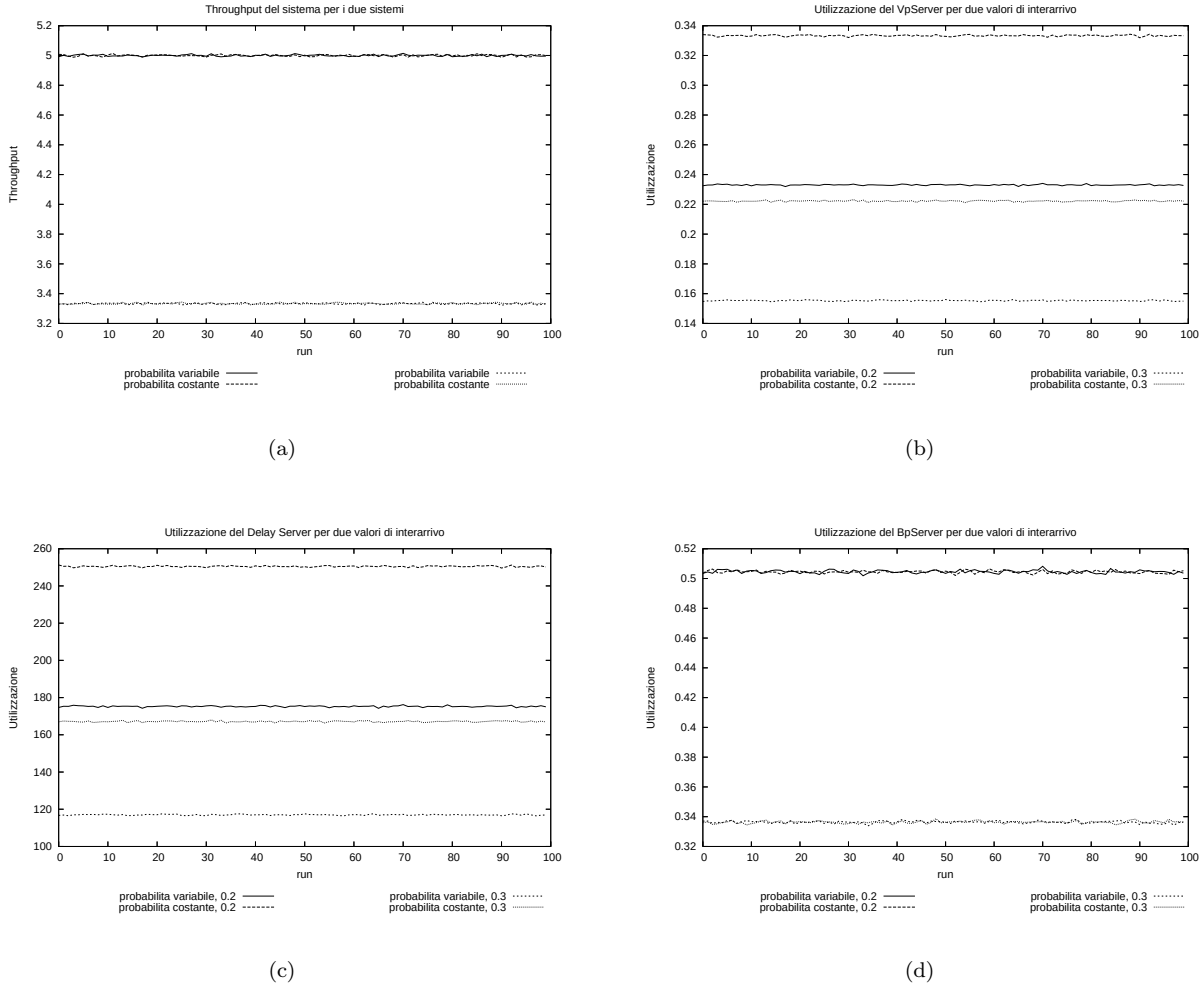


Figura 2.7: (a) Throughtput del sistema in condizioni di stabilità ed al variare della probabilità di diramazione. Probabilità costante indica che gli utenti usciti dal Delay Server hanno 70% delle possibilità di ritornare al VpServer mentre negli altri casi proseguono nel BpServer. Con probabilità variabile si intende che la possibilità di ritornare al VpServer dipende dal numero di visite effettuate secondo la funzione di figura 2.2. (b),(c) e (d) Utilizzazione dei tre serventi.

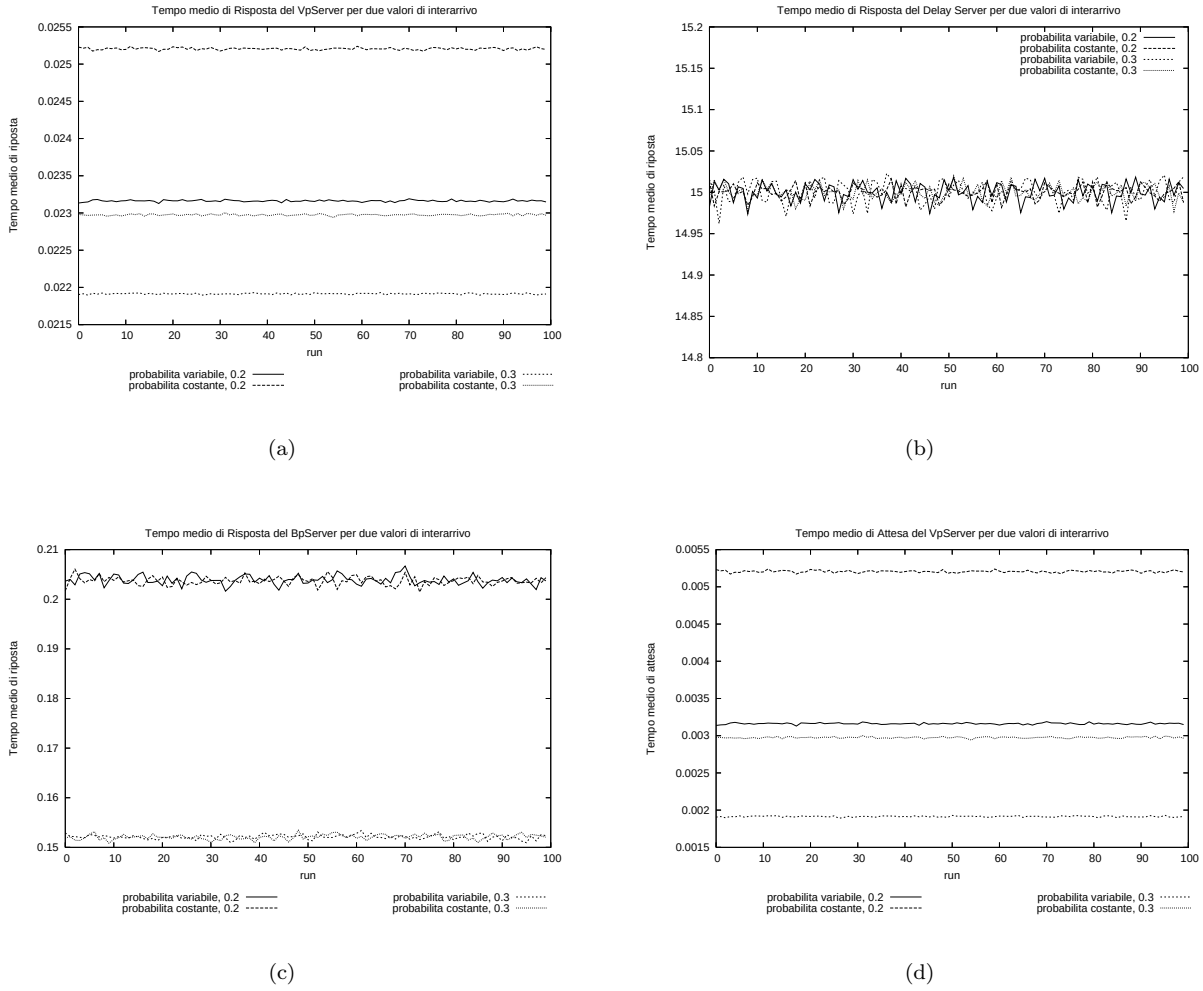


Figura 2.8: (a),(b) e (c) Tempi medi di risposta dei tre serventi. Solamente il tempo medio di risposta del VpServer è cambiato al variare della probabilità di diramazione. Questo valore sarà inferiore nel caso di diramazione variabile perché implica un numero inferiore di utenti nel nodo, quindi un tempo di attesa più basso. (d) Tempo medio di attesa del VpServer.

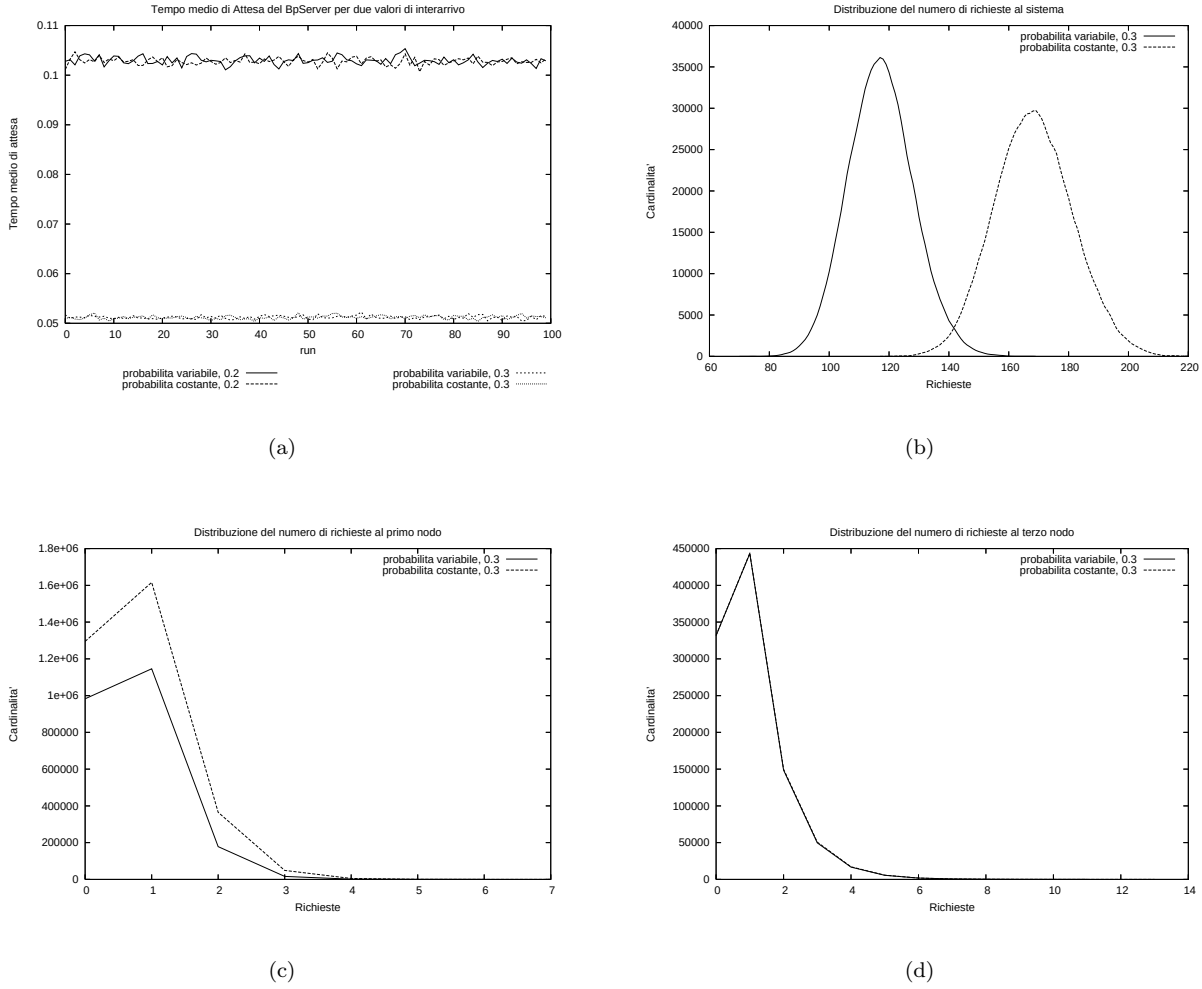


Figura 2.9: (a) Tempo medio di attesa del BpServer. (b),(c) e (d) Distribuzione delle richieste al sistema e agli altri due serventi. Il primo grafico testimonia come l'assunzione di una probabilità variabile porti ad uno spostamento della campana verso un valore medio inferiore. Il secondo grafico indica che la probabilità costante di diramazione porta all'aumento degli utenti che sono presenti nel VpServer.

Appendice A

Riepilogo Formule

In questa sezione viene inserita una lista di formule che sono state utilizzate per il calcolo degli indici di prestazione per un sistema classico M/M/1.

Utilizzazione	$\rho = \frac{\lambda}{\mu}$
Numero medio utenti:	$\overline{N} = \frac{\rho}{1-\rho}$
Response time medio:	$\overline{R} = \frac{1/\mu}{1-\rho}$
Waiting time medio:	$\overline{W} = \frac{\rho/\mu}{1-\rho}$
Lunghezza media della coda:	$\overline{Q} = \frac{\rho^2}{1-\rho}$

Tabella A.1: Formule utilizzate nella sezione 1.3

I parametri μ_1 e μ_2 di una distribuzione iperesponenziale H_2 possono essere approssimati usando una media data, un coefficiente di variazione campionato ed utilizzando le seguenti formule:

$$\mu_1 = \frac{1}{\overline{X}} \left[1 - \sqrt{\frac{q_2}{q_1} \frac{c_X^2 - 1}{2}} \right]^{-1}$$

e

$$\mu_2 = \frac{1}{\overline{X}} \left[1 - \sqrt{\frac{q_1}{q_2} \frac{c_X^2 - 1}{2}} \right]^{-1} .$$

Appendice B

Codice Sorgente

In questa sezione, viene riportato per velocità di consultazione, il codice sorgente delle classi più importanti che formano il simulatore. Il codice riguarda il simulatore modificato rispetto alle assunzioni (discipline di scheduling e distribuzioni del tempo di servizio) del secondo compito.

B.1 Main.java

```
public class Main {

    public static long[] seed_array=new long[12];
    public static boolean showdata=false;
    public static boolean attivaCampionatore=false;

    /** Creates a new instance of Main */
    public Main() {
    }
    public static void main(String[] args){
        Controller control;
        if (args.length>2){
            try {
                int i;
                for (i=0; i<12; i++){
                    seed_array[i]=Long.parseLong(args[i]);
                    //System.out.println("seme "+i+"="+seed_array[i]+"\n");
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
            //passando 'true' al costruttore del controller, specifico che i semi
            //non devono essere generati, ma utilizzati quelli passati come parametro.
            control=new Controller(true);
        }else{
            control=new Controller(false);
        }
        control.Await();
        System.exit(0);
    }
}
```

B.2 Controller.java

```
public class Controller extends SimulationProcess {

    /* uniform stream necessary to decide where the jobs "goes" after thinking */
    public static UniformStream uStream;

    /* first random number not to consider */
    public static final int START_RANDOM_STREAM=20;
    public static final double FASE_TRANSIENTE=4000.0;

    public static long nroJobArrivatiDallEsterno;
    public static long nroJobUscitiDalSitema;

    /* tempi di servizio dei 3 serventi utilizzati */
    public static Variance TempiServizioVPS=new Variance();
    public static Variance TempiServizioBPS=new Variance();
    public static Variance TempiServizioDS=new Variance();

    /* tempi di risposta T_q dei 3 serventi utilizzati */
    public static Variance TempiRispostaVPS=new Variance();
    public static Variance TempiRispostaBPS=new Variance();
    public static Variance TempiRispostaDS=new Variance();

    /* tempi di attesa T_w dei 2 serventi VPS e BPS */
    public static Variance TempiAttesaVPS=new Variance();
    public static Variance TempiAttesaBPS=new Variance();

    /* distribuzione del numero di utenti nel sistema*/
    public static Histogram distribuzioneUtentiSistema=new Histogram(500);
    public static Histogram distribuzioneUtentiVPS=new Histogram(500);
    public static Histogram distribuzioneUtentiBPS=new Histogram(500);

    public static long NumeroUtentiSitema=0;
    public static long numeroUtentiDS=0;
    public static long NumeroUtentiVPS=0;
    public static long NumeroUtentiBPS=0;

    public static TimeVariance UtentiSistema=new TimeVariance();

    public static TimeVariance UtentiNodo1=new TimeVariance();
    public static TimeVariance UtentiNodo2=new TimeVariance();
    public static TimeVariance UtentiNodo3=new TimeVariance();

    public static double totalServiceTimeVpS;
    public static double totalServiceTimeBpS;

    /* Delay Server Random Stream Parameters */
    private static final double serviceMeanDelayServer=15.0d;
    public static ExponentialStream serviceTimesDelayS;

    private double tempoMassimoDiSimulazione=150000.0d;

    /*nelle istruzioni successive, definisco gli elementi del sistema: arrivi, serventi e code*/
    public static Arrivals arr;
    public static ViewProductsServer vpServer;
    public static BuyProductsServer bpServer;
    public final static Queue<Job> codaViewProductsServer=new Queue<Job>();
    public final static Queue<Job> codaBuyProductsServer=new Queue<Job>(Main.seed_array[10],Main.seed_array[11]);
    public static Sampler camp;
    public static ResetCounters resettaContatori;
```

```
/* Array of Delay Server */
public static ArrayDS<DelayServer> listaDS=new ArrayDS<DelayServer>();


private static boolean semiDaFile;

/** Creates a new instance of Controller */
public Controller(boolean inizializzaConSemiDaInput) {
    semiDaFile=true;
    if (inizializzaConSemiDaInput==false){
        System.out.println("inizializzazione interna dei semi");
        semiDaFile=false;
        long seed1,seed2;
        seed1=transformIntoOdd(Calendar.getInstance().getTimeInMillis());
        seed2=seed1*78901;

        uStream=new UniformStream(0,1,START_RANDOM_STREAM,seed1,seed2);
        serviceTimesDelayS=new ExponentialStream(serviceMeanDelayServer,START_RANDOM_STREAM,
         transformIntoOdd(seed1*94032),seed2*57412);
    }else
        inizializzazioneSemi();
}

public void Await(){
    this.Resume();
    SimulationProcess.mainSuspend();
}
private void inizializzazioneSemi(){
    /* Ordine di prelievo semi:
     * 0) temi interarrivo dei clienti dall'esterno
     * 2) serviceTime VPS,
     * 4) serviceTime DS,
     * 6) serviceTime BPS,
     * 8) uniform stream per prob,
     * 10) random access per coda.
     */
    /* inizializzazione stream uniforme per probabilita' diramazione in DS */
    uStream=new UniformStream(0,1,START_RANDOM_STREAM,Main.seed_array[8],Main.seed_array[9]);
    /* inizializzatore service time DS stream */
    serviceTimesDelayS=new ExponentialStream(serviceMeanDelayServer,START_RANDOM_STREAM,
     Main.seed_array[4],Main.seed_array[5]);

}

private void generaSemiPerStepSuccessivo(){

    long tmp;
    File file=new File("semi_i.txt");
    try {
        file.createNewFile();
        FileOutputStream fos=new FileOutputStream(file,true);
        PrintStream ps=new PrintStream(fos);
        System.setOut(ps);

     /* Ordine di prelievo semi:
     * 0) temi interarrivo dei clienti dall'esterno
     * 2) serviceTime VPS,
     * 4) serviceTime DS,
     * 6) serviceTime BPS,
     * 8) uniform stream per prob,
     * 10) random access per coda.
```

```
     */
        System.out.println(transformIntoOdd(arr.interArrivalStream.MSeed));
        System.out.println(arr.interArrivalStream.LSeed);
        System.out.println(transformIntoOdd(vpServer.serviceTimesVpS.MSeed));
        System.out.println(vpServer.serviceTimesVpS.LSeed);
        System.out.println(transformIntoOdd(serviceTimesDelayS.MSeed));
        System.out.println(serviceTimesDelayS.LSeed);
        System.out.println(transformIntoOdd(bpServer.serviceTimesBpS.MSeed()));
        System.out.println(bpServer.serviceTimesBpS.LSeed());
        System.out.println(transformIntoOdd(uStream.MSeed));
        System.out.println(uStream.LSeed);
        System.out.println(transformIntoOdd(codaBuyProductsServer.us.MSeed));
        System.out.println(codaBuyProductsServer.us.LSeed);

        ps.close();
        fos.close();
    } catch (FileNotFoundException ex) {
        ex.printStackTrace();
    } catch (IOException ex) {
        ex.printStackTrace();
    }

}

public void run(){

    try{
        //istanziazione dei processi del sistema
        if (Main.attivaCampionatore==true){
            camp=new Sampler();
        }else{
            resettaContatori=new ResetCounters();
        }

        if (this.semiDaFile==true){
            arr=new Arrivals(Main.seed_array[0],Main.seed_array[1]);
            vpServer=new ViewProductsServer(Main.seed_array[2],Main.seed_array[3]);
            bpServer=new BuyProductsServer(Main.seed_array[6],Main.seed_array[7]);
        }else{
            arr=new Arrivals();
            vpServer=new ViewProductsServer();
            bpServer=new BuyProductsServer();
        }

        //attivazione dei processi
        arr.Activate(); //gli altri processi vengono attivati all'interno di Arrivals
        if (Main.attivaCampionatore==false){
            //se si attiva il campionatore, il progetto viene utilizzato per determinare la fase transiente.
            resettaContatori.Activate();
        }else{
            camp.Activate();
        }

        //inizio della simulazione
        Scheduler.startSimulation();
        Hold(tempoMassimoDiSimulazione);

        //fine simulazione
        Scheduler.stopSimulation();

        //terminazione processi del sistema
```

```
            arr.terminate();
            vpServer.terminate();
            bpServer.terminate();
            for (int i=0; i<this.listaDS.size();i++){
                this.listaDS.get(i).terminate();
            }
            if (Main.attivaCampionatore==true){
                camp.terminate();
            }else{
                resettaContatori.terminate();
            }
            if (Main.showdata==true){
                stampaConteggi();
            }
            generaSemiPerStepSuccessivo();
            salvaIndici();
            SimulationProcess.mainResume();
            Suspend();

        }catch(Exception ex){
            System.out.println("Controller Exception: "+ex.toString());
        }
    }

    public void salvaIndici(){
        File file=new File("indici.txt");
        double Teffettivo=tempoMassimoDiSimulazione-FASE_TRANSIENTE;
        try {
            PrintStream restoreSystemOut=System.out;

            file.createNewFile();
            FileOutputStream fos=new FileOutputStream(file,true);
            PrintStream ps=new PrintStream(fos);
            System.setOut(ps);

            System.out.println("X= "+(double)nroJobUscitiDalSitema/Teffettivo);

            System.out.println("U1= "+TempiServizioVPS.sum()/Teffettivo);
            System.out.println("U2= "+UtentiNodo2.mean());
            System.out.println("U3= "+TempiServizioBPS.sum()/Teffettivo);


            System.out.println("Rm1= "+TempiRispostaVPS.mean());
            System.out.println("Rm2= "+TempiServizioDS.mean());
            System.out.println("Rm3= "+TempiRispostaBPS.mean());

            System.out.println("Wm1= "+TempiAttesaVPS.mean());
            System.out.println("Wm3= "+TempiAttesaBPS.mean());

            System.out.println("Rv1= "+TempiRispostaVPS.variance());
            System.out.println("Rv2= "+TempiServizioDS.variance());
            System.out.println("Rv3= "+TempiRispostaBPS.variance());

            System.out.println("Wv1= "+TempiAttesaVPS.variance());
            System.out.println("Wv3= "+TempiAttesaBPS.variance());

            ps.close();
            fos.close();

            //scrivo su file le distribuzioni
            //distribuzioni del numero medio di utenti nel sistema
```

```
            file=new File("distrUtentiSistema.txt");
            file.createNewFile();
            fos=new FileOutputStream(file,true);
            ps=new PrintStream(fos);
            System.setOut(ps);
            this.distribuzioneUtentiSistema.print();
            ps.close();
            fos.close();
            //distribuzione del numero di utenti (richieste) al nodo 1
            file=new File("distrUtentiNodo1.txt");
            file.createNewFile();
            fos=new FileOutputStream(file,true);
            ps=new PrintStream(fos);
            System.setOut(ps);
            this.distribuzioneUtentiVPS.print();
            ps.close();
            fos.close();
            //distribuzione del numero di utenti (richieste) al nodo 3
            file=new File("distrUtentiNodo3.txt");
            file.createNewFile();
            fos=new FileOutputStream(file,true);
            ps=new PrintStream(fos);
            System.setOut(ps);
            this.distribuzioneUtentiBPS.print();
            ps.close();
            fos.close();

            System.setOut(restoreSystemOut);

        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public static long transformIntoOdd(long num){
        if (isEven(num) && (num-1>0))
            return (num-1);
        return num;
    }
}
```

B.3 Arrivals.java

```
public class Arrivals extends SimulationProcess{

    //mean inter arrival time
    private static final double meanInterArrivalTime=0.2;
    public ExponentialStream interArrivalStream;

    /** Creates a new instance of Arrivals */
    public Arrivals() {
        long seed1,seed2;
        seed1=Controller.transformIntoOdd(Calendar.getInstance().getTimeInMillis());
        seed2=seed1*81024;
        interArrivalStream=new ExponentialStream(meanInterArrivalTime,Controller.START_RANDOM_STREAM,seed1,seed2);
    }
    public Arrivals(long MGSeed, long LCGSeed) {
        interArrivalStream=new ExponentialStream(meanInterArrivalTime,Controller.START_RANDOM_STREAM,MGSeed,LCGSeed);
```

```
    }

    public void run(){
        try{
         Hold(interArrivalStream.getNumber());
        for (;;){

                //crea un nuovo job
                Job job=new Job(0,(int)Controller.nroJobArrivatiDallEsterno);

                Controller.nroJobArrivatiDallEsterno++;

                //accoda il job al nodo 1
                job.setArrivalTimeInQueue(CurrentTime());
                job.setTultimoIngressoCoda(CurrentTime());
                Controller.codaViewProductsServer.add(job);

                //quando c'e' un arrivo, il numero di utenti nel sistema aumenta di uno
                Controller.NumeroUtentiVPS++;
                Controller.UtentiNodo1.setValue(Controller.NumeroUtentiVPS);
                Controller.distribuzioneUtentiVPS.setValue(Controller.NumeroUtentiVPS);

                Controller.NumeroUtentiSitema++;
                Controller.UtentiSistema.setValue(Controller.NumeroUtentiSitema);
                Controller.distribuzioneUtentiSistema.setValue(Controller.NumeroUtentiSitema);


                //attiva il primo servente
                Controller.vpServer.Activate();
                Hold(interArrivalStream.getNumber());

         }
        }catch(Exception e){
                System.out.println("Arrivals Exception: " + e.toString());
        }
    }
}
```

B.4 VpServer.java

```
public class ViewProductsServer extends SimulationProcess{

    public static NormalStream serviceTimesVpS;
    //mean service time for this server
    private static final double meanServiceTimeVpS=20;
    private static final double varianceServiceTimeVpS=4;

    /** Creates a new instance of ViewProductsServer without parameters */
    public ViewProductsServer() {
        long seed1,seed2;
        seed1=Controller.transformIntoOdd(Calendar.getInstance().getTimeInMillis());
        seed2=seed1*25901;
        serviceTimesVpS=new NormalStream(meanServiceTimeVpS,varianceServiceTimeVpS,
         Controller.START_RANDOM_STREAM,seed1,seed2);
    }

    /** Creates a new instance of ViewProductsServer with seed parameters */
    public ViewProductsServer(long MGSeed, long LCGSeed) {
        serviceTimesVpS=new NormalStream(meanServiceTimeVpS,varianceServiceTimeVpS,
```

```
    Controller.START_RANDOM_STREAM, MGSeed,LCGSeed);
}
public void run(){
    double timeToHoldForService;

    double quanto;
    for (;;){
        try{
            //prelevo dalla coda un job
            Job job=Controller.codaViewProductsServer.poll();

            if (job==null){
                Passivate();
            }else{
                /* generazione tempo di servizio */
                if (job.getServiceTime()==0){
                    //si assegna ad un cliente un tempo di servizio secondo la distribuz.
                    timeToHoldForService=serviceTimesVpS.getNumber();
                    timeToHoldForService/=1000;
                    Controller.TempiServizioVPS.setValue(timeToHoldForService);
                    job.setServiceTime(timeToHoldForService);
                    job.setElaborazioneRimanente(timeToHoldForService);
                }

                timeToHoldForService=1.0/((1.0/meanServiceTimeVpS)*(Controller.codaViewProductsServer.size()+1));

                job.aggiuntiAtotaleAttesa((CurrentTime()-job.getTultimoIngressoCoda()));
                if (job.getElaborazioneRimanente()>timeToHoldForService){
                    //quanto pieno
                    quanto=job.getElaborazioneRimanente()-timeToHoldForService;
                    job.setElaborazioneRimanente(quanto);
                    Hold(timeToHoldForService);
                    job.setTultimoIngressoCoda(CurrentTime());
                    Controller.codaViewProductsServer.add(job);
                }else{
                    //quanto rimamente. Alla fine di questo hold, il cliente ha finito
                    //la visita e puo' passare al Delay Server
                    Hold(job.getElaborazioneRimanente());
                    job.setElaborazioneRimanente(0.0);

                    Controller.TempiAttesaVPS.setValue(job.getTotaleAttesa());
                    job.incrementaVisiteEff();
                    Controller.NumeroUtentiVPS--;
                    Controller.UtentiNodo1.setValue(Controller.NumeroUtentiVPS);
                    Controller.distribuzioneUtentiVPS.setValue(Controller.NumeroUtentiVPS);
                    Controller.TempiRispostaVPS.setValue(CurrentTime()-job.getArrivalTimeInQueue());
                    attivaDelayServer(job);
                }

            }

        }catch(Exception e){
            System.out.println("Vp Run Exception:"+e.toString());
            e.printStackTrace();
        }
    }
}

private void attivaDelayServer(Job job){
    boolean exitFlag=true;
    DelayServer d;
```

```
        try {
            Iterator<DelayServer> it=Controller.listaDS.iterator();
            while (it.hasNext() && exitFlag){
                //si cerca un DelayServer passivated
                d=it.next();
                if (d.passivated()){
                    exitFlag=false;
                    d.setJob(job);
                    Controller.numeroUtentiDS++;
                    Controller.UtentiNodo2.setValue(Controller.numeroUtentiDS);
                    d.Activate();
                }
            }
            if (exitFlag){
                d=new DelayServer(job);
                Controller.listaDS.add(d);
                Controller.numeroUtentiDS++;
                Controller.UtentiNodo2.setValue(Controller.numeroUtentiDS);

                d.Activate();
            }

        } catch (Exception ex){
            System.out.println("Vp Attiva Delay S Exception:"+ex.toString());
        }
    }
}
```

B.5 DelayServer.java

```
public class DelayServer extends SimulationProcess{

    private Job job;
    /** Creates a new instance of DelayServer */
    public DelayServer(Job job) {
        this.job=job;
    }
    public void setJob(Job job){
        this.job=job;
    }
    private double getLikelihood(int visite){
        if ((visite<=15) && (visite>=0)){
            return (4.0/150.0)*(double)visite+0.5;
        }else if ((visite>15) && (visite<=50)){
            return (-9.0/350.0)*(double)visite+(9.0/7.0);
        }else
            return 0.0;
    }
    public void run()  {
        for (;;){
            try{
                double timeToHoldForService=Controller.serviceTimesDelayS.getNumber();
                Hold(timeToHoldForService);
                Controller.TempiServizioDS.setValue(timeToHoldForService);

                double prob=Controller.uStream.getNumber();
                Job j=new Job(job.getNroVisiteEffettuate(),job.getIndex());

                j.setArrivalTimeInQueue(CurrentTime());
```

```
                j.setTultimoIngressoCoda(CurrentTime());

                if (prob<getLikelihood(j.getNroVisiteEffettuate())){
                    Controller.NumeroUtentiVPS++;
                    Controller.UtentiNodo1.setValue(Controller.NumeroUtentiVPS);
                    Controller.distribuzioneUtentiVPS.setValue(Controller.NumeroUtentiVPS);
                    Controller.codaViewProductsServer.add(j);
                }else{
                    Controller.codaBuyProductsServer.add(j);
                    Controller.NumeroUtentiBPS++;
                    Controller.UtentiNodo3.setValue(Controller.NumeroUtentiBPS);
                    Controller.distribuzioneUtentiBPS.setValue(Controller.NumeroUtentiBPS);
                }
                Controller.numeroUtentiDS--;
                Controller.UtentiNodo2.setValue(Controller.numeroUtentiDS);
                Controller.bpServer.Activate();
                Controller.vpServer.Activate();

                Passivate();
            }catch(Exception e){
                System.out.println("Delay S. Exception: " + e.toString());
            }
        }

    }
}
```

B.6 BpServer.java

```
public class BuyProductsServer extends SimulationProcess{


    //stream di numeri distribuiti secondo ...
    public static HyperExp2Stream serviceTimesBpS;
    //mean service time for this server
    private static final double meanServiceTimeBpS_1=0.098184995;
    private static final double meanServiceTimeBpS_2=0.102722506;

    /** Creates a new instance of BuyProductsServer without parameters */
    public BuyProductsServer() {
        long seed1,seed2;
        seed1=Controller.transformIntoOdd(Calendar.getInstance().getTimeInMillis());
        seed2=seed1*98765;
        serviceTimesBpS=new HyperExp2Stream(0.6,meanServiceTimeBpS_1,meanServiceTimeBpS_2,
         Controller.START_RANDOM_STREAM,seed1,seed2);
    }

    /** Creates a new instance of BuyProductsServer with seed parameters */
    public BuyProductsServer(long MGSeed, long LCGSeed) {
        serviceTimesBpS=new HyperExp2Stream(0.6,meanServiceTimeBpS_1,meanServiceTimeBpS_2,
         Controller.START_RANDOM_STREAM,MGSeed,LCGSeed);
    }

    public void run(){
        for (;;){
            try{
                Job job=Controller.codaBuyProductsServer.pollRandom();

                if (job==null){
```

```
                Passivate();
            }else{
                double timeToHoldForService=serviceTimesBpS.getNumber();
                Controller.TempiAttesaBPS.setValue(CurrentTime()-job.getArrivalTimeInQueue()); //T_w


                Hold(timeToHoldForService);


                //quando un utente esce dal sistema, il conteggio deve essere decrementato.
                Controller.NumeroUtentiSitema--;
                Controller.UtentiSistema.setValue(Controller.NumeroUtentiSitema);

                Controller.TempiServizioBPS.setValue(timeToHoldForService);
                Controller.TempiRispostaBPS.setValue(CurrentTime()-job.getArrivalTimeInQueue());
                Controller.nroJobUscitiDalSitema++;
                Controller.NumeroUtentiBPS--;
                Controller.UtentiNodo3.setValue(Controller.NumeroUtentiBPS);
                Controller.distribuzioneUtentiBPS.setValue(Controller.NumeroUtentiBPS);

                Controller.distribuzioneUtentiSistema.setValue(Controller.NumeroUtentiSitema);
            }
        }catch(Exception e){
            System.out.println("Buy Product Exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

}
```

B.7 Sampler.java

```
public class Sampler extends SimulationProcess{

    private static final double HoldTime=10.0d;
    public static String data="";
    File file=null;
    public static PrintStream ps=null;
    FileOutputStream fos;
    /** Creates a new instance of Sampler */
    public Sampler() {
        try{
            file=new File("Sampler.txt");
            file.createNewFile();
            fos=new FileOutputStream(file,true);
            ps=new PrintStream(fos);
        }catch(Exception e){
            System.out.println("Sampler init exception: " + e.toString());
        }
    }

    public void run(){
        for(;;){
            try{
                data=data+(Controller.NumeroUtentiSitema)+"\n";
                Hold(HoldTime);
                if (data.length()>=2000){
```

```
                    ps.print(data);
                    data="";
                }
            }catch(Exception e){
                System.out.println("Sampler Exception: "+e.toString());
                System.exit(1);
            }
        }
    }
}
```

B.8 Job.java

```
public class Job {

    private int index;
    private double serviceTime=0.0;
    private double responseTime;
    private double arrivalTimeInQueue;
    private int nroVisiteEffettuate;
    private double quanto;
    private double elaborazioneRimanente;
    private double TingressoCoda;
    private double totaleAttesa=0.0;

    /** Creates a new instance of Job */
    public Job(int nroVisiteEffettuate, int index) {
        this.index=index;
        this.serviceTime=0.0;
        this.totaleAttesa=0.0;
        this.nroVisiteEffettuate=nroVisiteEffettuate;
    }
    public Job(int nroVisiteEffettuate, double TingressoCoda, double totaleAttesa,
            double elaborazioneRimanente, double arrivalTimeInQueue, double serviceTime){
        this.serviceTime=serviceTime;
        this.arrivalTimeInQueue=arrivalTimeInQueue;
        this.elaborazioneRimanente=elaborazioneRimanente;
        this.TingressoCoda=TingressoCoda;
        this.totaleAttesa=totaleAttesa;
        this.nroVisiteEffettuate=nroVisiteEffettuate;
    }

    //metodi per l'accesso ai campi privati della classe
    public int getIndex(){
        return this.index;
    }
    public void setIndex(int index){
        this.index=index;
    }
    public double getTultimoIngressoCoda(){
        return this.TingressoCoda;
    }

    public void setTultimoIngressoCoda(double t){
        this.TingressoCoda=t;
    }

    public double getTotaleAttesa(){
        return this.totaleAttesa;
```

```
    }
    public void aggiuntiAtotaleAttesa(double aggiungi){
        this.totaleAttesa+=aggiungi;
    }
    public void setTotaleAttesa(double attesa){
        this.totaleAttesa=attesa;
    }
    public double getElaborazioneRimanente(){
        return this.elaborazioneRimanente;
    }
    public void setElaborazioneRimanente(double rim){
        this.elaborazioneRimanente=rim;
    }
    public int getNroVisiteEffettuate(){
        return this.nroVisiteEffettuate;
    }
    public void incrementaVisiteEff(){
        this.nroVisiteEffettuate++;
    }
    public void setNroVisiteEffettuate(int NewVisite){
        this.nroVisiteEffettuate=NewVisite;
    }

    public double getServiceTime(){
        return this.serviceTime;
    }
    public void setServiceTime(double serviceTime){
        this.serviceTime=serviceTime;
    }
    public double getArrivalTimeInQueue(){
        return this.arrivalTimeInQueue;
    }
    public void setArrivalTimeInQueue(double arrivalTimeInQueue){
        this.arrivalTimeInQueue=arrivalTimeInQueue;
    }
}
```

B.9 Queue.java

```
public class Queue<E> extends LinkedList<E> {
    private int nroElementiCircolati;
    public static UniformStream us;
    Random rand;
    /** Creates a new instance of Queue */
    public Queue() {

    }
    public Queue(long MGSeed, long LCGSeed){
        us=new UniformStream(0.0,1.0,Controller.START_RANDOM_STREAM,MGSeed,LCGSeed);
    }
    synchronized public boolean add (E o){
        this.nroElementiCircolati++;
        return super.add(o);
    }

    synchronized public E poll(){
        //Retrieves and removes the head (first element) of this list.
        return super.poll();
    }
```

```
    synchronized public E get(int i){
        return (E)super.get(i);
    }
    synchronized public E pollRandom(){
        //estrai un elemento a caso dalla lista

        int nelementi=this.size();

        if (nelementi==0)
            return null;
        else{
            int p;
            try {
                p = (int) Math.floor(us.getNumber()*((double)nelementi-1.0));
                E job=(E) this.get(p);
                this.remove(p);
                return job;
            } catch (ArithmeticException ex) {
                ex.printStackTrace();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
           return null;
        }
    }
    synchronized public int size(){
        return super.size();
    }
    synchronized public int getNumeroElementiCircolati(){
        return this.nroElementiCircolati;
    }
}
```

B.10 ResetCounters.java

```
public class ResetCounters extends SimulationProcess {

    /** Creates a new instance of ResetCounters */
    public ResetCounters() {

    }
    public void run(){

        try {
            Hold(Controller.FASE_TRANSIENTE);
            Controller.nroJobUscitiDalSitema=0;
            Controller.nroJobUscitiDalSitema=0;

            Controller.UtentiNodo1.reset();
            Controller.UtentiNodo2.reset();
            Controller.UtentiNodo3.reset();

            Controller.TempiServizioVPS.reset();
            Controller.TempiServizioBPS.reset();
            Controller.TempiServizioDS.reset();

            /* tempi di risposta T_q dei 3 serventi utilizzati */
            Controller.TempiRispostaVPS.reset();
            Controller.TempiRispostaBPS.reset();
```

```
            Controller.TempiRispostaDS.reset();

            /* tempi di attesa T_w dei 2 serventi VPS e BPS */
            Controller.TempiAttesaVPS.reset();
            Controller.TempiAttesaBPS.reset();

            //Controller.UtentiSistema.reset();
            Controller.distribuzioneUtentiSistema.reset();
            Passivate();
        } catch (SimulationException ex) {
            ex.printStackTrace();
        } catch (RestartException ex) {
            ex.printStackTrace();
        }
    }

}
```

Bibliografia

[1] Simonetta Balsamo. Dispense del corso di prestazioni e affidabilità sistemi e modelli di valutazione di sistemi. `http://www.dsi.unive.it/~balsamo`.

[2] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Trivedi. *Queueing Networks and Markov Chains*. John Wiley and Sons, 1998.

[3] Universitá degli Studi di Torino. Quaderni didattici del dipartimento di matematica. `http://www.dm.unito.it/quadernididattici/2001d.html/`.

[4] Lazowska Edward D., Zahorjan John, Graham G. Scott, and C. Sevcik Kenneth. *Quantitative System Performance Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[5] Raj Jain. *Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling*. John Wiley and Sons, 1991.

[6] L Kleinrock. *Queueing Systems*. John Wiley and Sons, 1975.

[7] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGrawHill, 2001.

[8] University of Newcastle upon Tyne. Javasim home page. `http://javasim.ncl.ac.uk/`.