FACOLTÀ DI INGEGNERIA INFORMATICA

CORSO DI MODELLI DI PRESTAZIONI DI SISTEMI E RETI

Prof.ssa Vittoria De Nitto Personé

Anno Accademico 2012/2013

Relazione Progetto

Serena Mastrogiacomo **serena.mastrogiacomo@gmail.com**
Emanuele Paracone **emanuele.paracone@gmail.com**

INTRODUZIONE

La presente relazione si prefigge lo scopo di descrivere il progetto previsto al termine del corso di Modelli di Prestazioni di Sistemi e Reti. Il progetto riguarda la risoluzione di un modello analitico per lo studio delle prestazioni di un'applicazione multi-tier.

CONTENUTI

- Nel primo capitolo verranno descritti gli obiettivi del lavoro , la descrizione del sistema in esame e le specifiche del progetto;

- Nel secondo capitolo viene fornita una panoramica sui diversi modelli esistenti per lo sviluppo di applicazioni multi-tier e vengono discussi i vantaggi e gli svantaggi delle varie soluzioni presenti in letteratura; infine vengono discussi i vantaggi della soluzione con introduzione di un componente (**CAC**) per il controllo sulle ammissioni delle sessioni.

- Nel terzo capitolo riportiamo l'implementazione del sistema e alcuni commenti relativi alle scelte modellistiche effettuate

- Nel quarto capitolo, infine, verrà riportata l'analisi dei risultati ottenuti, sia in forma numerica che in forma grafica, opportunatamente commentati.

In Appendice 1 riportiamo il listing del codice prodotto.

CAPITOLO 1: Obiettivi

L'obiettivo del progetto è la realizzazione e la risoluzione di un modello analitico per lo studio delle prestazioni di un'applicazione multi-tier. Nell'ambito dei servizi internet sono sempre più numerosi quelli dedicati a e-business: solitamente sono strutturati come applicazioni web multi-tier costituite da un web server, un application server ed un back-end server. La richiesta di servizio da parte di un client viene processata e servita attraverso l'interazione dei vari livelli: ogni livello i fornisce funzionalità al livello i-1 e utilizza le funzionalità del livello i+1 per portare a termine le proprie attività. Nello sviluppo di applicazioni di questo tipo occorre fare attenzione non solo a garantire i livelli di prestazione concordati attraverso le SLA, ma anche a ridurre lo spreco di risorse al minimo possibile. Dalla prospettiva dell'utente, è importante avere la possibilità di usufruire del servizio secondo tempistiche ragionevoli e soprattutto senza interruzioni, mentre per il provider del servizio risulta fondamentale fornire il servizio nel rispetto di quanto stabilito nello SLA, onde evitare sanzioni e con il minor spreco di risorse possibile. Questo obiettivo viene posto per ridurre i costi e di conseguenza per massimizzare i profitti. Come descritto in [1], nello studio delle prestazioni di un servizio costituito da un'applicazione multi-tier, risultano fondamentali le attività di:

(i) capacity provisioning per la pianificazione delle capacità dei singoli tier in modo tale da rispettare i vincoli specificati negli SLA;

(ii) performance prediction per la stima del tempo di risposta del sistema o per la stima di altri indici di prestazioni locali e glibali;

(iii) application configuration che prevede diverse configurazioni del sistema per conseguire diversi obiettivi;

(iv) bottleneck identification and tuning per l'identificazione dei componenti che costituiscono un rallentamento non trascurabile nel sistema;

(v) request policing per individuare la migliore politica di ammissione delle richieste.

Tra le attività impiegate nell'ambito della simulazione conosciamo anche il *workload*

forecasting per la previsione e pianificazione degli aggiornamenti dei vari componenti in base al carico di sistema, per mantenere inalterati i livelli prestazionali e l'attività di *system tuning* per ottenere ed impostare i valori ottimi dei parametri di sistema, come valori soglia e discipline di servizio. Al fine di conseguire questi task diventa quindi necessario realizzare un modello del sistema che sia il più conforme possibile a quello reale. Una volta realizzato, il modello va risolto con uno dei metodi analitici proposti in letteratura per ottenere uno studio accurato degli indici di prestazione locali e globali.

1.1 Modelli a reti di code

L'obiettivo del lavoro riguarda la modellazione tramite un modello a reti di code chiuso di un sistema relativo ad un'applicazione multi-tier costituita da un web server, un application server ed un back-end database e la determinazione di alcuni indici di prestazione locali e globali tramite l'applicazione di due principali metodi risolutivi. In questo contesto la teoria delle code si propone lo sviluppo di modelli che descrivano fenomeni di attesa, in cui le richieste degli utenti vengono servite da centri di servizio, le cui caratteristiche variano a seconda del dominio del problema; in particolare vengono qui analizzati i modelli a rete di code per la descrizione di una rete di centri di servizi. La soluzione di tali modelli è possibile per via analitica attraverso due approcci:

- Markoviano
- Forma Prodotto

L'approccio Forma Prodotto è l'approccio richiesto per la risoluzione del problema in esame sia con algoritmo Gordon&Newell sia con metodo di risoluzione MVA. Gli approcci verranno brevemente descritti nel capitolo 4 per descrivere in maniera completa il lavoro realizzato, mentre per una descrizione dettagliata si faccia riferimento a [2].

1.2 Descrizione del sistema

Si consideri il problema della gestione efficiente dei servizi Internet, importante sia a livello di produttività personale che di business. In questo contesto, si consideri una tipica applicazione multi-tier che consiste di un web server, un application server ed un database (backend) server. Ad esempio, nell'ambito delle applicazioni di e-business, l'accesso a un servizio Web avviene sotto forma di sessione che consiste di molte richieste individuali. Ordinare un prodotto via Internet, coinvolge ulteriori richieste quali selezionare un prodotto, fornire informazioni sulla spedizione, gestire il pagamento e finalmente ricevere una conferma. Dal punto di vista dell'utente, la misura reale di prestazione del web server, *è la sua capacità di elaborare l'intera sequenza di richieste necessarie per completare la transazione.*

Allo scopo, si consideri un meccanismo di controllo dell'ammissione (CAC) volto ad accettare una nuova sessione solo quando il sistema ha sufficiente capacità per elaborare tutte le future richieste collegate a quella sessione. In altri termini, solo quando il sistema può garantire il completamento con successo della sessione.

Si consideri il modello a rete di code del sistema oggetto dello studio (figura 1.1). La richiesta per una nuova sessione, una volta accettata dal sistema, compie un accesso al frontend ed al backend server prima di tornare al client. Si noti che il web server e l'application server risiedono usualmente sullo stesso server fisico, perciò il sistema può essere modellato, ad eccezione dei centri necessari per modellare i client, da due centri a coda per il frontend ed il backend server rispettivamente. Una volta che la richiesta ritorna al client, questo spende un "think time" prima di generare una nuova richiesta. Una sessione viene completata dopo che il client ha generato una serie di richieste.

Il meccanismo CAC viene modellato mediante la seguente estensione al modello base:

- un vincolo di capacità di sistema S sulla popolazione massima ammessa nel sistema (FEserver +BEserver). Per poter distinguere tra nuove richieste di connessione e richieste appartenenti a sessioni già ammesse, si introduce il Client 2. Perciò, le richieste provenienti da Client 1 modellano l'instaurazione di nuove sessioni, mentre le richieste provenienti da Client 2 modellano richieste successive di sessioni già in corso;

- una probabilità di rifiuto delle richieste quando il sistema raggiunge la capacità S.

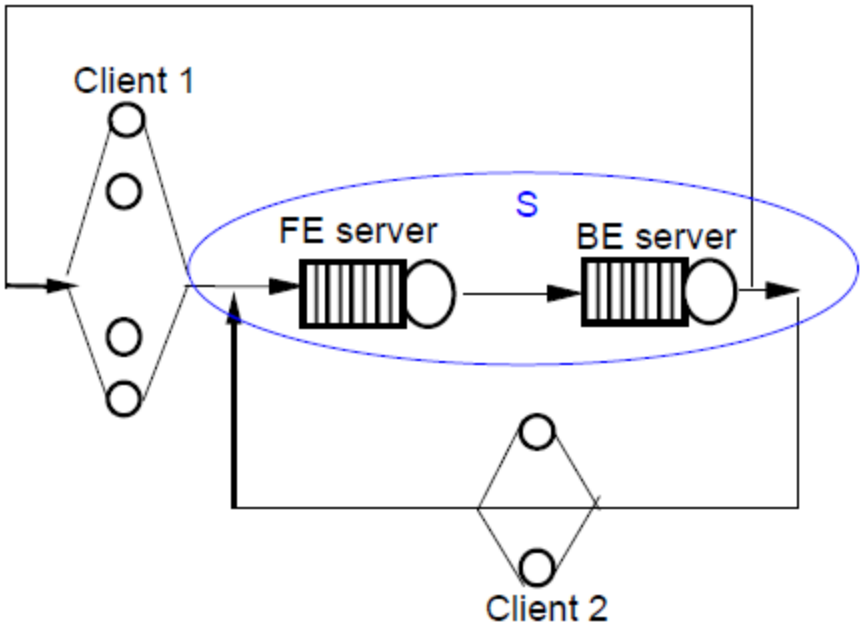

Figura 1.1

1.3 Specifiche

Considerando i modelli descritti nella precedente sezione,si risolva il modello (figura 1.1) in Forma Prodotto e si determini la probabilità di saturazione come segue:

psat(N, S) = Prob{ nFE+ nBE ≥ S}

Si consideri un nuovo modello come illustrato in figura 1.2. Il modello include due centri aggiuntivi per modellare la probabilità di rifiuto della richiesta nei periodi di saturazione, entrambi di parametro esponenziale di media 0.05 sec. Si definiscano le nuove probabilità di routing da C_i a C_{i_reject}, i=1,2, come segue:

$p_{c1,c1\_rj} = p_{sat}(N, S1)$
$p_{c2,c2\_rj} = p_{sat}(N, S2)$

Per la soluzione del nuovo modello si utilizzi MVA.

Il sistema deve soddisfare il seguente requisito di QoS:
$p_{ci,ci\_rj} \leq 0.22$
dove p_{ci,ci_rj}, i=1,2, rappresentano le probabilità di reject ai due client. Si progetti un meccanismo di tuning del sistema che, agendo sulle due soglie S1 e S2, minimizzi il tempo medio di risposta del sistema trclient1, rispetto a Client 1, al variare del carico N.

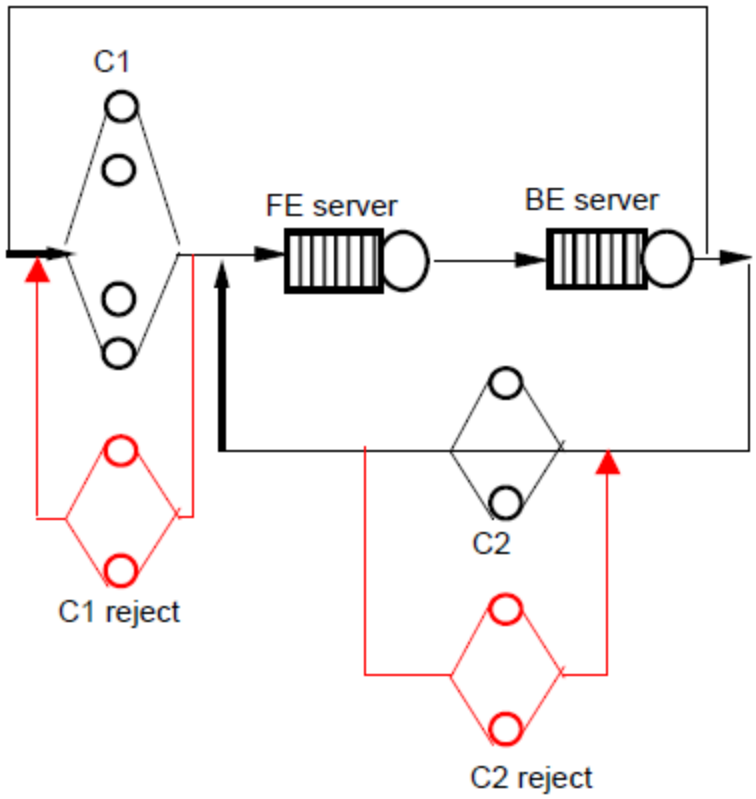

Figura 1.2

Si considerino i seguenti parametri:
Numero di job: N=50

1.3.1 CARATTERISTICHE DEI CENTRI

Client 1 e Client 2: IS esponenziali di media 7 sec
FEserver: FIFO esponenziale di media 0.3 sec
BEserver: FIFO esponenziale di media 0.08 sec
Probabilità di routing da BEserver a Client 1: $p_{be,Client1} = 1/21 = 0.048$
Probabilità di routing da BEserver a Client 2: $p_{be,Client2} = 1 - p_{be,Client1} = 0.952$

CAPITOLO 2: Scelte modellistiche

In questo capitolo vogliamo fornire una panoramica sui diversi modelli esistenti per lo sviluppo di applicazioni multi-tier opportunatamente discussi in [1]. Un primo metodo è attraverso la fusione dei diversi tier in un unico tier, modellando quindi con una sola coda il centro con più vincoli, il collo di bottiglia o l'intero sistema. Il sistema risultante corrisponde ad un impianto aperto, con una sorgente che genera job ed un unico centro che modella tutta l'applicazione. Nonostante la facilità di realizzazione del modello e il vantaggio dovuto alla presenza di numerosi risultati analitici già presenti in letteratura, tale soluzione offre però diversi svantaggi tra cui l'impossibilità di effettuare capacity planning, la difficoltà nella cattura degli effetti del caching tra i vari livelli e infine problematiche relative alle sessioni e al controllo di amministrazione. L'accorpamento inoltre rende impreciso il calcolo degli indici di prestazione locali e globali. Un'altra soluzione possibile è quella di modellare ogni tier come un centro a coda e in questo modo è possibile risolvere i problemi riscontrati della soluzione precedente: ogni tier può essere analizzato in termini di performance prediction, tier configuration e system tuning. Nel progetto proposto, tale soluzione prevede l'uso di un modello composto da 3 code, ognuna delle quali modella un application tier ed il sottostante server che la esegue (figura 2.1).

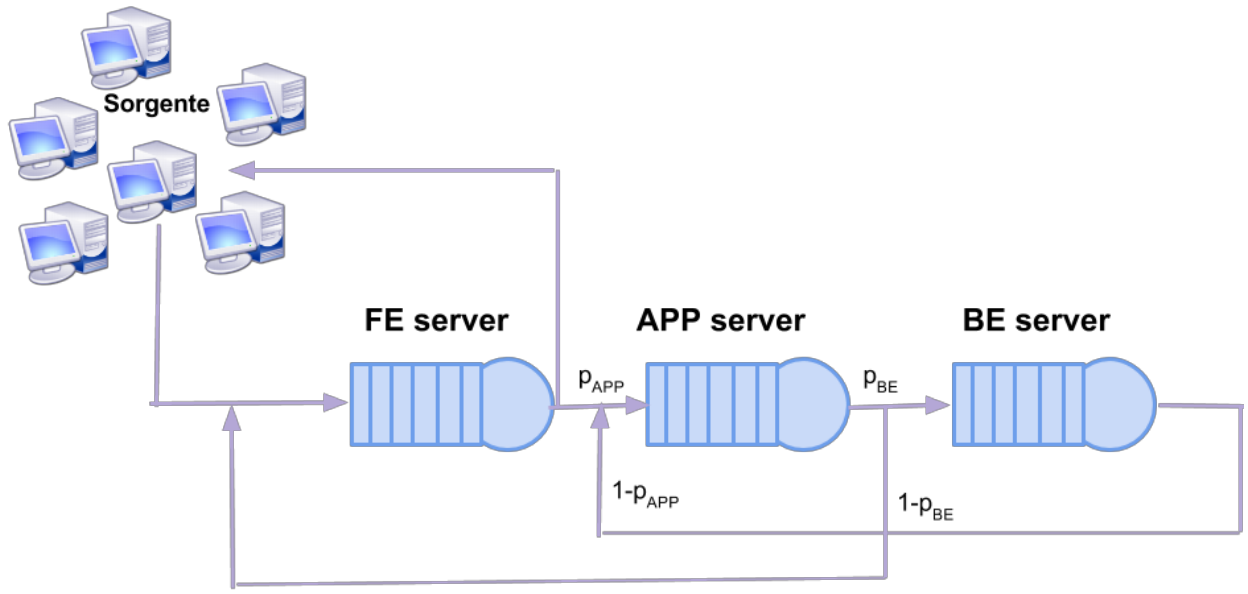

Figura 2.1 - Three Tier

Ogni tier i all'arrivo di una richiesta innesca una serie di richieste al tier i+1; le probabilità di routing del tier i+1 modellano sia la possibilità di proseguire il lavoro con ulteriori richieste al tier i+2, sia quella di terminare e ritornare al tier i che aveva generato la richiesta. Attraverso tali probabilità di routing è ora possibile modellare e catturare gli effetti del caching ai vari livelli, funzionalità che con la presenza di una singola coda non era realizzabile: una richiesta pervenuta al tier i dal tier i-1, in caso di cache hit al tier i, ritorna immediatamente a quest'ultimo con una opportuna transizione. Al termine del processamento, la richiesta torna al tier M (FE Server) e da qui poi alla sorgente, indicando che la richiesta è stata portata a termine. In genere il web server e l'application server risiedono sulla stessa macchina, pertanto il sistema può essere modellato come un two-tier in cui il primo tier modella il complesso costituito da Front-end ed Application Server ed il secondo modella il Back-end Server come si osserva in figura 2.2.

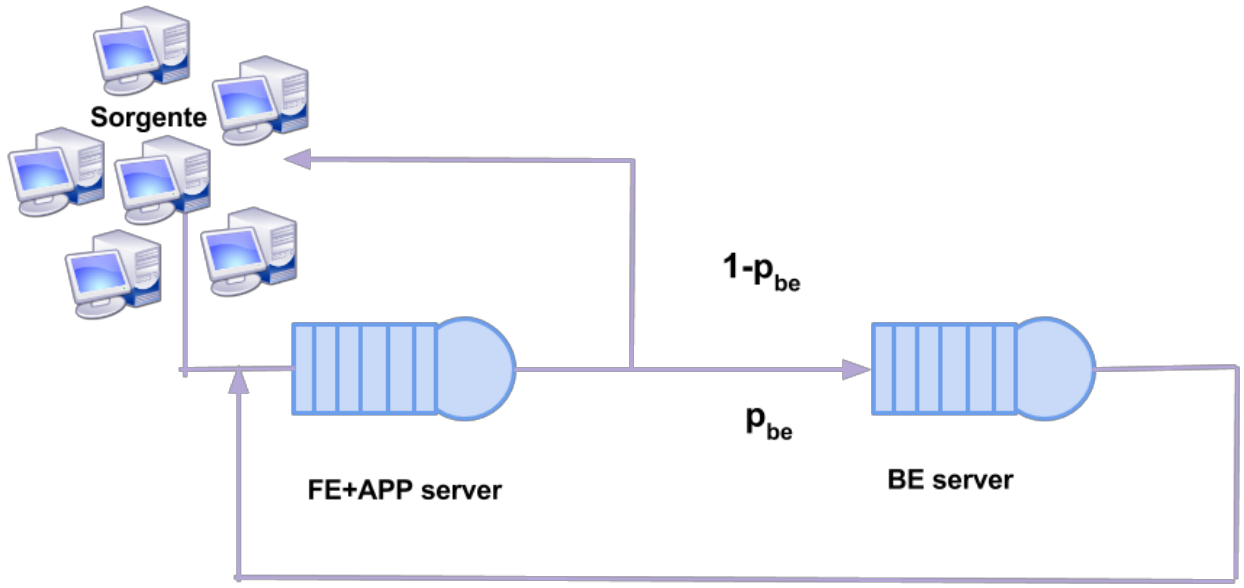

Figura 2.2- Two Tier

Non essendo necessario modellare esplicitamente l'interazione di ritorno tra BE server e FE+APP server, il modello può essere ulteriormente semplificato come mostrato nella figura seguente (2.3).

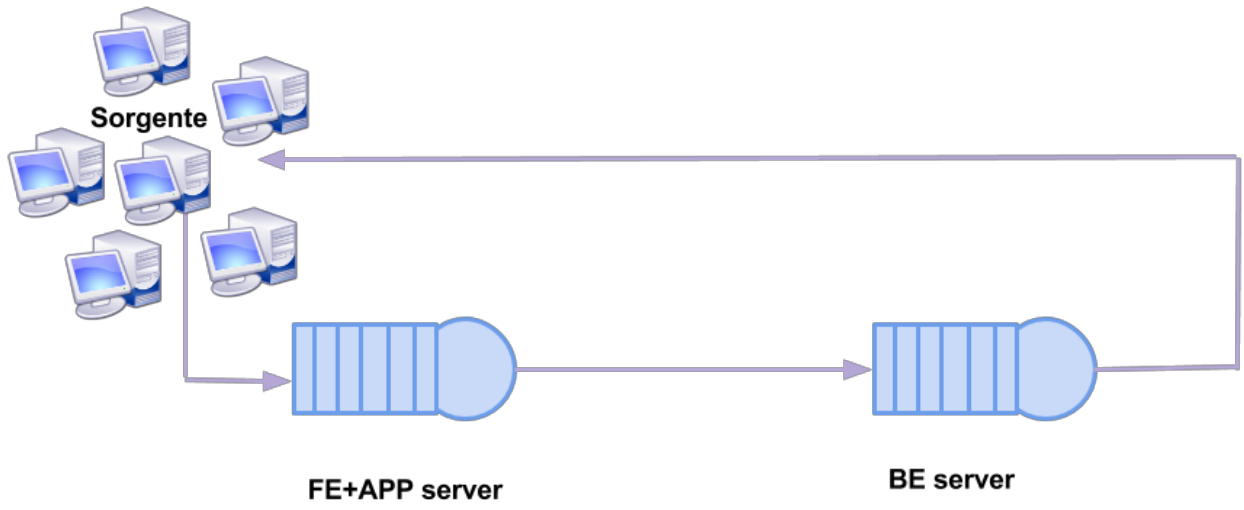

Figura 2.3-Two Tier senza interazione di ritorno tra BE server e FE+APP server

In generale l'accesso ad un servizio web avviene sotto forma di sessione, nel corso della quale vengono effettuate una o più richieste individuali intervallate da think-time dell'utente. Ad esempio, ordinare un prodotto via internet coinvolge una sequenza di richieste da parte dei client come selezionare un prodotto, fornire informazioni sull'indirizzo per la spedizione, effettuare il pagamento ed infine ricevere la conferma dell'avvenuta transazione. Il modello rappresentato in figura 2.3 permette tuttavia di modellare soltanto il tempo medio di risposta della singola richiesta all'interno di una sessione. Dovendo catturare non solo questo aspetto ma anche la lunga durata di tali sessioni, nonché le iterazioni contenute al loro interno, occorre modificare il modello per consentire la cattura delle caratteristiche appena elencate. Come suggerito nell'articolo [1] è necessario aggiungere alla rete un sottosistema che modelli le sessioni attive dell'applicazione. Questo aspetto viene modellato con l'aggiunta di un infinite server (Client 1), il quale popola la rete con le richieste degli utenti e forma un loop sul FE Server, come mostrato in figura 2.4.

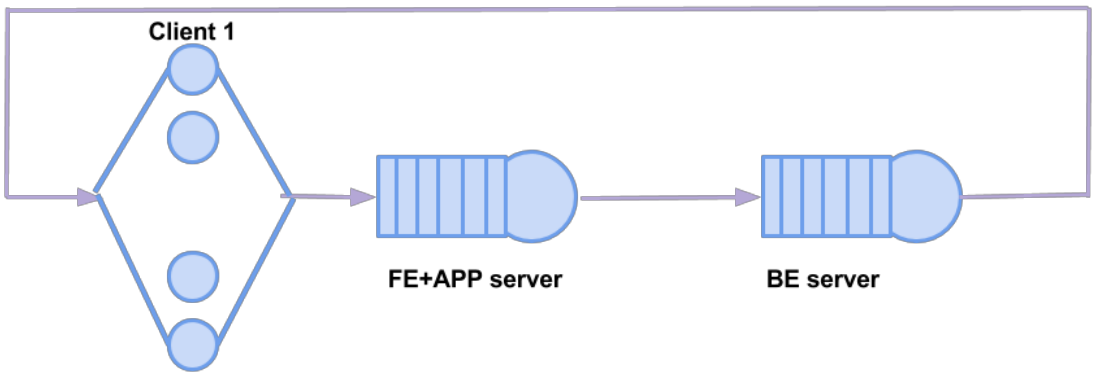

Figura 2.4- Two Tier con sessioni

Per poter distinguere tra nuove richieste e richieste appartenenti a sessioni preesistenti, si introduce nel modello il client 2 come si osserva nella figura seguente (2.5).

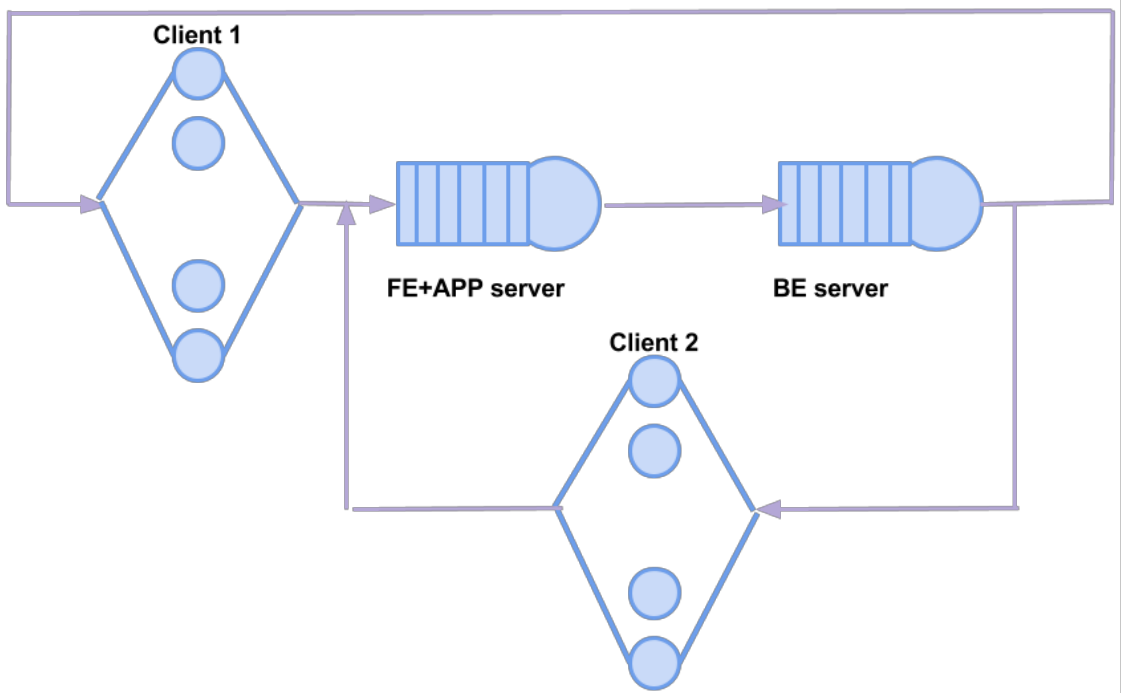

Figura 2.5- Two Tier con sessioni distinte

Le richieste provenienti dal client 1 modellano l'instaurazione di nuove sessioni mentre quelle provenienti dal client 2 modellano le successive richieste delle sessioni già in corso. Durante il ciclo di vita della sessione, la singola richiesta passa al FE+APP server e al BE server per poi tornare ai client (Client 1 o al Client 2). Successivamente, all'interno del client viene speso un think time prima che venga generata un'ulteriore richiesta. Si assume che ogni sessione preesistente utilizzi un servente del Client 2.

Le richieste riguardanti una data sessione vengono processate secondo il normale flusso che va dal FE+APP server al BE server; se contiene altre richieste allora si passa dal BE Server al Client 2, altrimenti si ritorna al Client 1. Il tempo medio di servizio del Client 2 modella il think time speso dall'utente; in questo modo viene preservata l'indipendenza tra il tempo medio dell'utente e il tempo per servire la richiesta da parte dell'applicazione. Il numero di sessioni servite può variare, poiché, nel corso del funzionamento dell'impianto, ci sono sessioni che iniziano e sessioni che terminano.

Tale modello riesce a calcolare il tempo medio di risposta per un numero N di sessioni simultanee. Generalmente le applicazioni web subiscono improvvisi picchi di traffico; ciò comporta due tipi di problematiche: da una parte risulta impossibile gestire contemporaneamente un numero infinito di client, dall'altra, se anche ciò fosse realizzabile, non si riuscirebbe a rispettare i vincoli imposti negli SLA.

Una possibile soluzione per la gestione di tali picchi consiste nell'introdurre un componente (**CAC**) per il controllo sulle ammissioni delle sessioni. Tale controllore, in caso di picco di traffico, deve essere in grado di selezionare un sottoinsieme delle sessioni che il sistema è in grado di completare: la scelta delle sessioni da accettare, come descritto in [1], deve avvenire in modo tale da riuscire a ottimizzare una metrica desiderata. Nel sistema in figura 2.6 ogni tier è in grado di servire un numero illimitato di sessioni simultanee; in realtà, per quanto appena spiegato, occorre impostare un valore soglia S, altrimenti non si riuscirebbe a modellare correttamente il comportamento dell'applicazione al superamento di tale limite. Si potrebbero interporre dei CAC tra i vari tier del sistema; il CAC del tier i può: eliminare le sessioni in eccesso e affidarsi ad un meccanismo di timeout del tier i-1 per il recupero delle sessioni eliminate oppure eliminare le sessioni in eccesso e segnalarne la rimozione al tier i-1 che può poi gestire opportunamente. In entrambi i casi il tier i-1 può liberamente decidere di ripetere la richiesta al tier i o di notificare l'eliminazione al tier i-2; tale notifica prosegue fino al tier M che provvede a propagarla fino al Client. Per modellare questo meccanismo, si introducono delle ulteriori transizioni, una per ogni tier; tra il tier i-1 e il tier i si colloca una diramazione che conduce ad un infinite server e da qui di nuovo al Client: una richiesta intraprende l'uno o l'altro cammino con una certa probabilità , simulando il rifiuto da parte del CAC.

Il tempo medio di servizio t_{i_reject} modella il ritardo medio dovuto alla perdita, come ad esempio il tempo necessario allo scadere di un timeout. In tal modo si può effettuare una distinzione tra le richieste processate e quelle scartate, in quanto le seconde sono soggette a ulteriori ritardi dovuti ai sottosistemi t_{i_reject} per la propagazione al client.

Per quanto riguarda il sistema in figura 2.6 , dato che non occorre un controllo di ammissione al livello dei singoli tier, si è scelto di eseguirlo sulla totalità dell'impianto. Il meccanismo CAC viene quindi modellato tramite un vincolo sul numero massimo di sessioni simultanee (S) nel complesso FE+APP+BE server ed una probabilità di rifiuto delle richieste al raggiungimento di S. Per modellare tale probabilità di rifiuto sono stati aggiunti due centri IS, uno per ogni client; ogni richiesta del client i viene rifiutata con una certa probabilità di rifiuto p_{ci_rj} o prosegue verso il FE+APP server con probabilità $1-p_{ci\_rj}$. Con l'utilizzo di tali probabilità di routing ed i relativi ritardi introdotti (think time dei server di drop), si modella senza perdita di generalità il rifiuto di richieste e gli annessi meccanismi di recupero, come timeout e segnalazione esplicita al tier precedente.

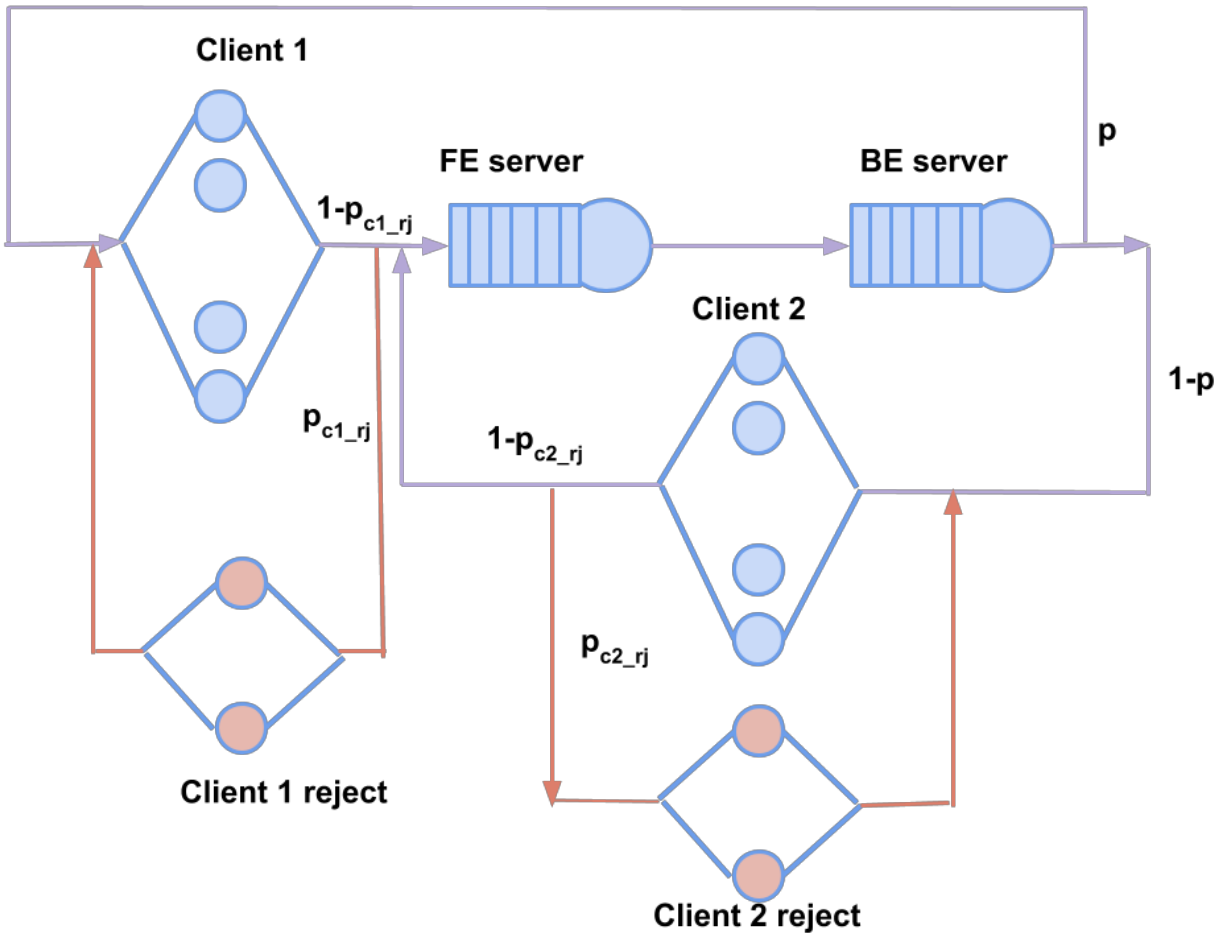

Figura 2.6 Two-Tier con CAC

CAPITOLO 3: Implementazione

Il codice per il calcolo degli indici che saranno discussi e commentati nel capitolo relativo alll'analisi dei risultati è stato implementato utilizzando Python v2.7.5+ come linguaggio di programmazione ed Eclipse Kepler con plugin pyDev come ambiente di sviluppo.
Per la realizzazione dei grafici sono state installate le seguenti librerie:

- per i grafici
    - la libreria matplotlib.pyplot
    - il framework mpl_toolkits.mplot3d
- per la serializzazione dei dati
    - la libreria di sistema cPickle

Descriviamo ora i passi effettuati per la risoluzione dei modelli in esame. Il codice sorgente è riportato in appendice ed è organizzato in tre componenti principali:

- main.py
- gordonNewell.py
- MVA.py

3.1 Risoluzione del primo modello

ALGORITMO GORDON & NEWELL

Per quanto riguarda la risoluzione del primo modello e la relativa implementazione è stato sufficiente seguire fedelmente i passi dell'algoritmo Gordon e Newell e per i calcoli più complessi utilizzare la suite open source SAGE. In questa sezione introduciamo brevemente i procedimenti svolti; per visionare il relativo codice sorgente fare riferimento alla classe `GordonNewell()` riportata nell'Appendice A.1.

GENERAZIONE DEGLI STATI

Il primo passo nell'applicazione della soluzione in forma prodotto con algoritmo di risoluzione Gordon e Newell , è stato quello di generare tutti gli stati in cui il sistema può trovarsi. Essendo quella in esame una rete chiusa, non esiste una sorgente esterna e si assume una popolazione fissa di N utenti (nel nostro caso N= 50). Nel nostro modello risultante si aveva una rete con N = 50 job e M = 4 centri. Lo spazio degli stati è espresso come in formula :

$$\left\{ E = (n_1, ..., n_i, ...n_M), \sum_{i=1}^{M} n_i = N, \forall n_i \geq 0 \right\}$$

è stato ottenuto attraverso la generazione di tutte le combinazioni possibili di N job (o utenti) negli M centri (`generaStati(self, index, jobNum))`.
La cardinalità dello spazio degli stati è:

$$N_s = \frac{(N+M-1)!}{N!(M-1)!}$$

Nel nostro caso **N_s=23426**. La generazione degli stati e stata gestita applicando la risoluzione con *ordine lessicografico inverso*: si basa sull'idea che i centri nella nostra rete vengono riempiti in passi successivi con il numero massimo di job che possono ospitare, distribuendo i rimanenti nei centri restanti. L'operazione viene ripetuta per tutti i job interni ad ogni centro, pertanto la prima sequenza di stati ottinuti è la seguente: [50,0,0,0], [49,1,0,0], [48,1,1,0], etc .

GENERAZIONE PROBABILITÀ DI STATO

Il numero di utenti N che circola nella rete, segue una specifica matrice di routing P = [p_{ij}], con $1 \leq i, j \leq M$ la quale è stata determinata banalmente a partire dalle caratteristiche dei centri riportate nel capitolo 1 sezione 1.3.1. Dalla matrice di routing P si è impostato il sistema per la determinazione dei valori dei throughput relativi $\bar{y} = \bar{y} P$ dove, come precedentemente introdotto, la grandezza y_i conincide con il rapporto tra visite v_{ij}. Per la risoluzione del sistema abbiamo utilizzato la suite open source SAGE .

CALCOLO DELLA COSTANTE DI NORMALIZZAZIONE

Il passo successivo e stato quello per il calcolo della *costante di normalizzazione* G(N). Questo perché nell'algoritmo Gordon&Newell tale costante compare nella formula per il calcolo delle probabilità di stato:

$$\pi(n_1, n_2, ..., n_M) = \frac{1}{G(N)} \prod_{i=1}^{M} \frac{x_i^{n_i}}{\beta_i(n_i)}$$

dove:

- $x_i = \frac{y_i}{\mu_i}$
- $G(N) = \sum_{\forall n \in E} \prod_{i=1}^{M} \frac{x_i^{n_i}}{\beta_i(n_i)}$
- $\beta_i(n_i) = n_i!\ se\ n_i \leq m_i \quad e\ \ \beta_i(n_i) = m_i! m_i^{n_i - m_i}\ se\ n_i > m_i$

Infine procediamo con il calcolo degli indici prestazionali che verranno opportunamente graficati e commentati nella sezione relativa all'analisi dei risultati.

3.2 Risoluzione del secondo modello

Per la risoluzione del secondo modello è stata necessaria un'analisi dei requisiti più approfondita, al fine di rispondere fedelmente all'obiettivo definito sul tempo di risposta del sistema sperimentato dai client di tipo 1. Da detta analisi dei requisiti è stato derivato un modello di supporto con lo scopo di parametrizzare correttamente il modello originale. In questa sezione illustriamo dapprima l'analisi dei requisiti giustificando la necessità del modello di supporto, per poi esporre una descrizione qualitativa di quest'ultimo.

3.2.1 Analisi dei requisiti

L'obiettivo definito dalle specifiche del progetto si prefigge di minimizzare il tempo di risposta del sistema sperimentato dai **Client 1** manipolando le due soglie **S1** e **S2** in funzione del numero di Job all'interno del sistema. Dette soglie definiscono il carico massimo imponibile all'insieme dei due serventi (Front End e Back End server) da parte delle due tipologie di richieste (quelle provenenti rispettivamente dal **Client 1**, necessarie ad instaurare una nuova sessione, e quelle provenienti da **Client 2**, che modellano gli accessi dovuti a sessioni esistenti). In altri termini, vogliamo minimizzare la durata media di una sessione completa, ovvero del tempo che intercorre tra il momento in cui ne viene richiesta l'istanziazione fino al suo completamento.

Osservando il modello, alla luce di quanto detto notiamo:

a. di non poter considerare come completamento di sessione i job che “tornano” presso il **Client 1** a causa della rejection imposta dal vincolo **S1** sul carico del sistema;

b. di dover tenere conto del ritardo introdotto dalle possibili rejection della prima connessione, comprensivo del tempo di elaborazione della rejection (ovvero il tempo di esecuzione di **Client 1 rej**) e del nuovo tempo di think presso il **Client 1**.

Tali considerazioni ci inducono ad adottare un modello di supporto **SM** (Support Model) da cui sia possibile derivare correttamente la durata media di una sessione in funzione delle soglie **S1** e **S2** e del numero di jobs presenti nel sistema. Utilizziamo dunque il modello di supporto per implementare il meccanismo di tuning che permetta di minimizzare il tempo medio di sessione, adottando conseguentemente le soglie ottenute per parametrizzare e risolvere il modello originale.

3.2.2 Il modello di supporto SM

Per quanto desunto dall’analisi dei requisiti (3.2.1), il modello di supporto assume la seguente forma:

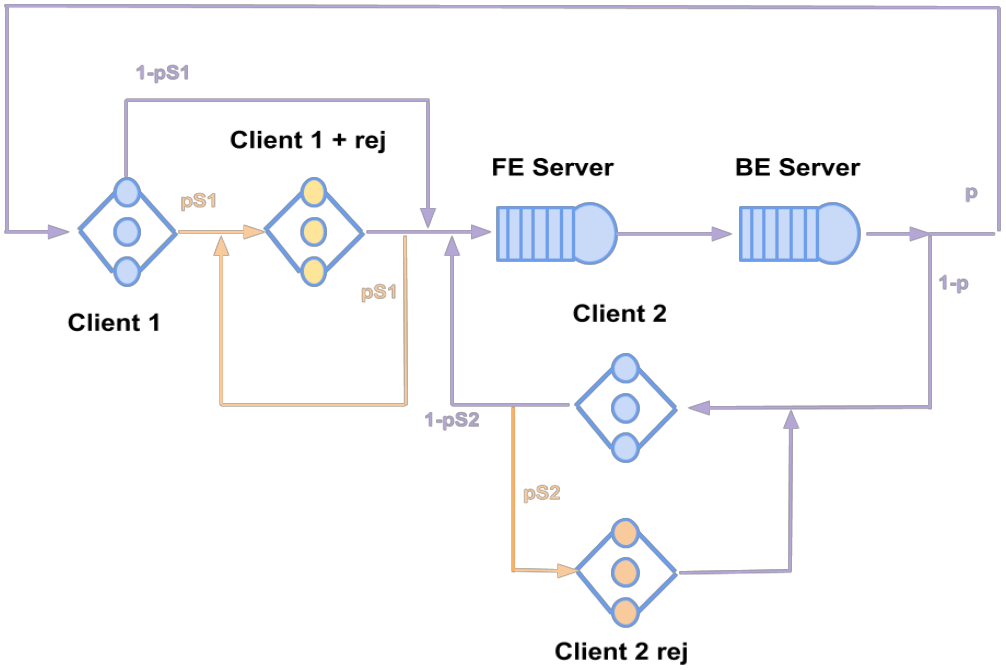

Figura 3.1-Modello SM

Tale modello impone il seguente routing al flusso dei jobs:

- una richiesta proveniente dal **Client 1** accede al sistema server con probabilità 1-$\mathbf{p_{S1}}$ (dove $\mathbf{p_{S1}}$ è la probabilità di rejection imposta dalla soglia **S1**);

- una richiesta proveniente dal **Client 1** accede al centro **Client 1+rej** con probabilità $\mathbf{p_{S1}}$;

- una richiesta proveniente dal **Client 1+rej** accede al sistema server con probabilità 1-$\mathbf{p_{S1}}$;

- una richiesta proveniente dal **Client 1+rej** fa ritorno al centro **Client 1+rej** con probabilità $\mathbf{p_{S1}}$;

- solo job provenienti dal **BE Server** possono fare ritorno a **Client 1**.

Confrontando il nuovo modello con quello originale, si nota la rimozione del centro **Client** $\mathbf{1_{rej}}$ e l'aggiunta di un centro **Client 1+rej**, il cui tempo medio di servizio è pari alla somma del tempo medio necessario a elaborare la rejection (ovvero il tempo medio di servizio del centro **Client1**$_{\mathbf{rej}}$) e del tempo medio di think presso **Client 1**.
In formule:

$$E[t_{s\ client1+rej}] = E[t_{s\ client\ 1}] + E[t_{s\ client\ 1_{rej}}]$$

In pratica, il modello di supporto trasforma il parallelo tra **Client 1** e **Client** $\mathbf{1_{rej}}$ nella serie **Client 1** - **Client 1+rej** .

3.2.3 Soluzione del modello

Per il soddisfacimento delle specifiche in esame (sezione 1.3) e alla luce di quanto osservato nell'analisi dei requisiti relativa al modello in questione (paragrafo 3.2.1), abbiamo realizzato un programma che realizza lo scopo in due fasi:

1. ricerca dei valori delle soglie S1 e S2 che, al variare del numero di jobs N, minimizzino il tempo di risposta del sistema visto dal Client 1 attraverso l'applicazione dell'algoritmo Mean Value Analisys (MVA) al modello di supporto;

2. calcolo degli indici locali e globali del sistema attraverso l'applicazione di MVA al modello originale parametrizzato con i valori delle soglie S1 e S2 trovati al passo precedente e relativi ad uno specifico numero N di jobs nel sistema.

ALGORITMO MVA

Come per l'algoritmo Gordon & Newell vogliamo dare un breve accenno ai passi eseguiti per l'applicazione dell'algoritmo MVA. Il codice che lo implementa è contenuto nella classe `MVA()` riportata in appendice A.3. Tale algoritmo rende possibile ottenere gli indici di prestazione locali e globali senza passare per il calcolo della costante di normalizzazione G(N), soluzione possibile tramite il *teorema degli arrivi*, che permette di derivare il tempo medio di risposta di un utente ad un centro come somma del tempo medio di servizio e del tempo medio di completamento del servizio per tutti gli utenti in coda al momento del suo arrivo. In generale MVA è da preferire per reti con centri a capacità fissa e centri di tipo IS. La classe MVA() implementa la ricorrenza:

$$\lambda_j(N) = \frac{N}{\sum_{I=1}^{M} v_i E(t_i(N))}$$

per j=1,....,M e con $v_i = \frac{y_i}{y_j}$

$$E(n_j(N)) = \lambda_j(N) v_j E(t_j(N))$$

$$E(t_j(N)) = E(t_j)(1 + E(n_j(N-1))$$

Tale ricorrenza consente di calcolare gli indici locali della rete con M centri ed N utenti in tempo polinomiale O(NM) più gli $O(M^3)$ passi per risolvere il sistema y=yP che fornisce i rapporti v_i.

Per il calcolo del tempo di risposta vale:

$E(t_j(N)) = E(t_{s_j})$ per centri IS

$E(t_j(N)) = E(t_{s_j})(1 + E(n_j(N-1))$ per centri con coda

Si ricorda che $n_j(N)$ è il numero istantaneo di utenti al centro i in una rete con N utenti totali, e $t_j(N)$ il tempo di risposta di detto centro. il teorema afferma che il tempo medio di risposta $E(t_j(N))$ visto da un utente che arrivi al centro i in una rete con N utenti, è pari al suo tempo medio di servizio $E(t_{sj})$ più quello degli $E(n_j(N-1))$ utenti che in media lo precederebbero in una rete con N-1 utenti. I risultati ottenuti con questo teorema sono commentati nella sezione dell'analisi dei risultati.

CAPITOLO 4: Analisi dei risultati

In questo capitolo saranno presentati i risultati derivati dalla risoluzione dei due impianti definiti nel capitolo delle specifiche (Sezione 1.3). Relativamente al primo impianto, verrà data un'illustrazione degli indici locali e globali derivati dalla sua soluzione ottenuta mediante l'algoritmo in forma prodotto di Gordon Newell.

Maggiore attenzione verrà dedicata alla soluzione del secondo impianto, ottenuta sempre tramite un approccio in forma prodotto ma attraverso l'adozione dell'algoritmo Mean Value Analisys (MVA), con un focus particolare sulla variazione del tempo di risposta del sistema rispetto al tentativo da parte di un job proveniente da Client 1 di instaurare e completare una sessione.

4.1 Soluzione del primo impianto

Come anticipato più volte nel presente lavoro, il primo modello di impianto è stato risolto tramite l'algoritmo in forma prodotto di Gordon Newell. Nel capitolo 3 (sezione 3.2) è stato descritto il workflow della nostra implementazione, che, seguendo pedissequamente la descrizione dell'algoritmo teorico, riteniamo sia superfluo approfondire ulteriormente.

Ci concentriamo pertanto nella sola illustrazione e spiegazione degli indici locali e globali ottenuti per il modello di impianto quando nel sistema sono presenti 50 utenti, facendo mostra dei risultati corollari riguardo alla probabilità di rifiuto di un job in funzione della soglia S definita sulla popolazione presente presso i centri **Fe Server + Be Server.**

4.1.1 Risultati

Di seguito sono riportati i dati ottenuti attraverso la soluzione del primo modello di impianto via algoritmo in forma prodotto Gordon Newell.

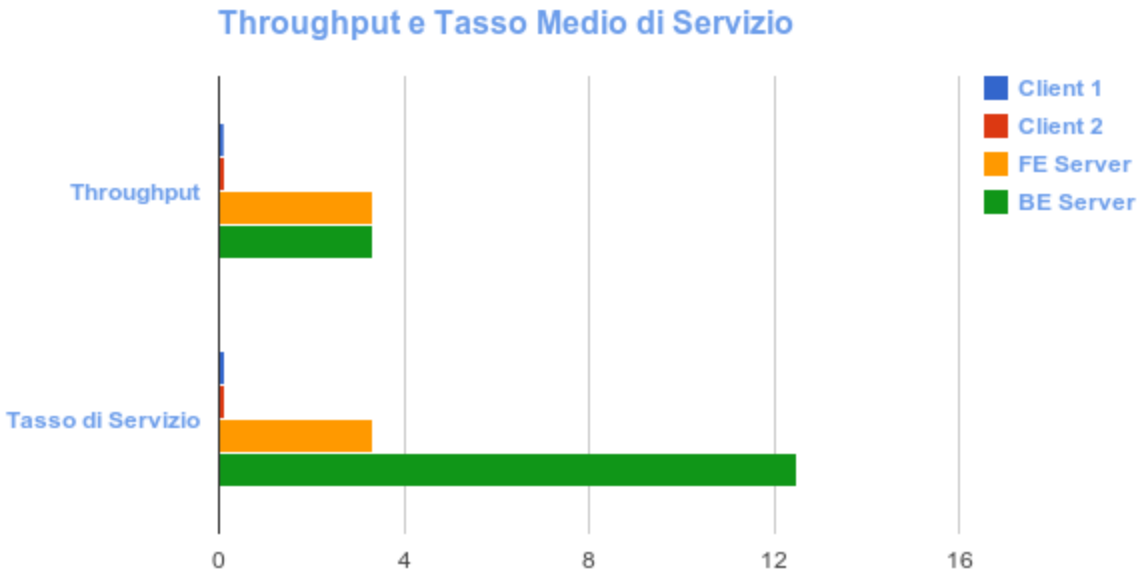

Figura 4.1

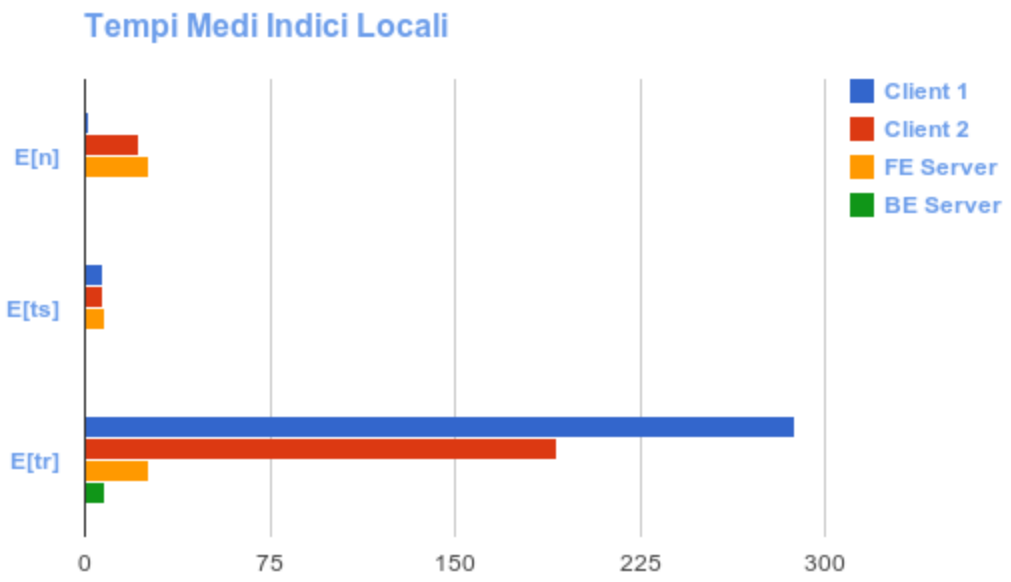

Figura 4.2

	Client 1	Client 2	FE Server	BE Server
Throughput	0.096245715794 62717907	0.14285714282494 874072	3.33332998587726 336837	3.33332998867726 937320
Tasso di Servizio	0.142857142857 14284921	0.14285714285714 284921	3.33333333333333 348136	12.5000000000000 0000000
Fattore di utilizzazione	-	-	0.99999899576317 896610	0.26666639909418 154986
$E[n]$	1.119998918979 51897590	22.2133110269613 8515284	26.3030544930459 4489490	0.36363556101322 219716
$E[t_q]$	7.0	7.0	7.89092427227048 975880	0.10909077776530 606840
$E[t_r]$	287.4589255782 5698077067	191.382870495480 74492122	26.3030544930459 4489490	8.39304077256530 867146

Tabella 4.1

4.1.2 Throughput e Tassi di Servizio Locali

Com'era intuibile, il throughput dei due server è lo stesso, in quanto subiscono lo stesso flusso di utenti da eseguire. Il tasso medio di servizio, invece, cambia radicalmente tra i due centri, in quanto il tempo medio di servizio del **BE Server** (0.08 sec) è nettamente inferiore rispetto a quello che caratterizza il **FE Server** (0.3 sec).

Consultando la tabella ci accorgiamo inoltre che il sistema server si trova in una situazione di effettiva congestione, in quanto il **FE Server** lavora con un fattore di utilizzazione in pratica pari ad 1.

4.1.3 Popolazioni Medie

La popolazione del sistema, quando sono presenti 50 jobs, si distribuisce prevalentemente tra il **FE Server** e il **Client 2**:

- l'accumulo di jobs presso FE Server è facilmente spiegabile a partire dal fattore di utilizzazione praticamente pari a 1: la coda del centro è sempre piena, in quanto il tasso di arrivi è pari al tasso di smaltimento;
- presso Client 2 i jobs sperimentano il ritardo dovuto al tempo di think imposto dall'utente che prende visione dell'output interno ad una sessione: tale tempo è mediamente lungo (7 sec) rispetto all'ordine di grandezza dei tempi di servizio dei server (decimi di secondo).

4.1.4 Tempo medio di risposta dell'impianto

Il tempo medio di risposta dell'impianto sperimentato dai singoli centri varia sensibilmente in funzione delle probabilità di routing. Ne consegue che i jobs provenienti dal centro con probabilità di routing più sfavorevoli, ovvero il **Client 1**, faranno ritorno allo stesso centro mediamente più tardi rispetto a quanto farebbero presso gli altri centri.

4.1.5 Probabilità di Soglia

Poiché l'algoritmo in forma prodotto Gorodon Newell prevede il calcolo delle probabilità stazionarie di stato per ogni possibile stato della forma $(n_1, n_2, n_{3,} n_4)$ t.c.

$n_i \in [0,50] \wedge \sum_{i=1}^{4} n_i = 50$ (considerando che il nostro modello prevede 4 centri e 50 jobs), abbiamo potuto calcolare le probabilità di rifiuto di un job che intende accedere al sistema server (ovvero al **FE Server**) in funzione della soglia di sbarramento S, calcolata come la somma degli utenti presenti ad un dato istante presso **FE Server** e **BE Server**.

In formula, abbiamo calcolato:

$$P(N_{FE\ Server} + N_{BE\ Server} \leq S) = \sum_{j=0}^{States\ Number} (\pi(n_1, n_2, n_3, n_4)\ t.c\ (n_2 + n_3 \leq S_i))$$

al variare di S_i tra i valori 1 e 50.
Le popolazioni presso i centri **Fe Server** e **Be Server** in funzione del numero di utenti presenti nel sistema sono illustrati nella figura seguente:

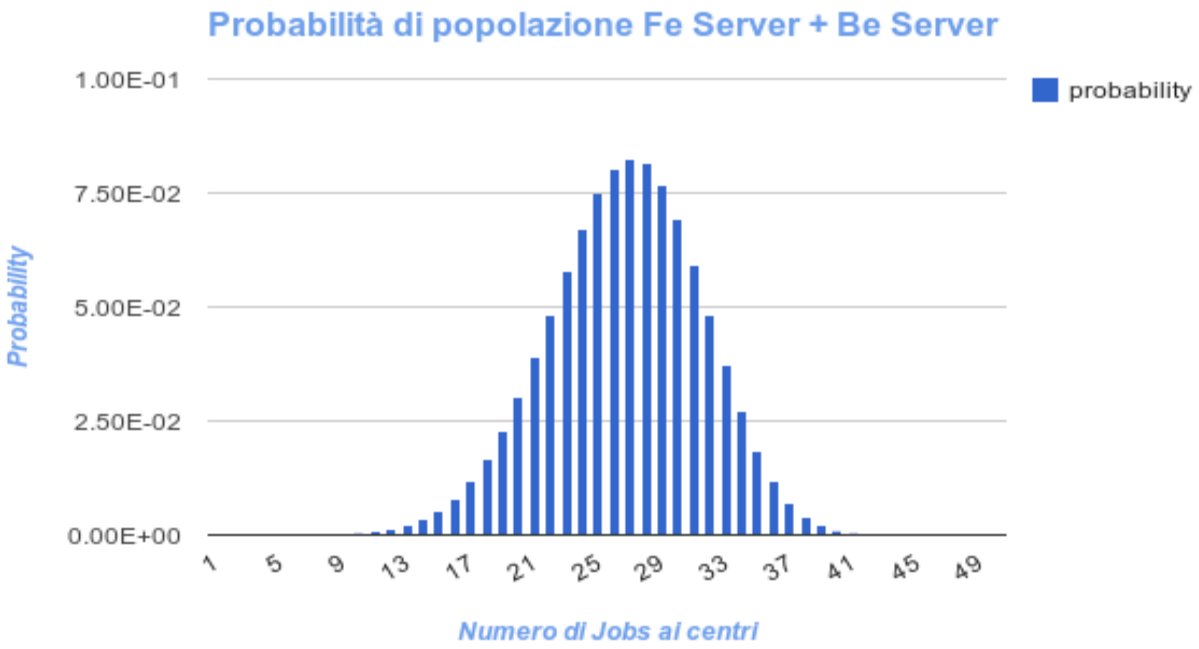

Figura 4.3

La probabilità di rifiuto sperimentata da un job che intende accedere al sistema dei server quando nel sistema sono presenti 50 jobs segue invece il seguente l'andamento:

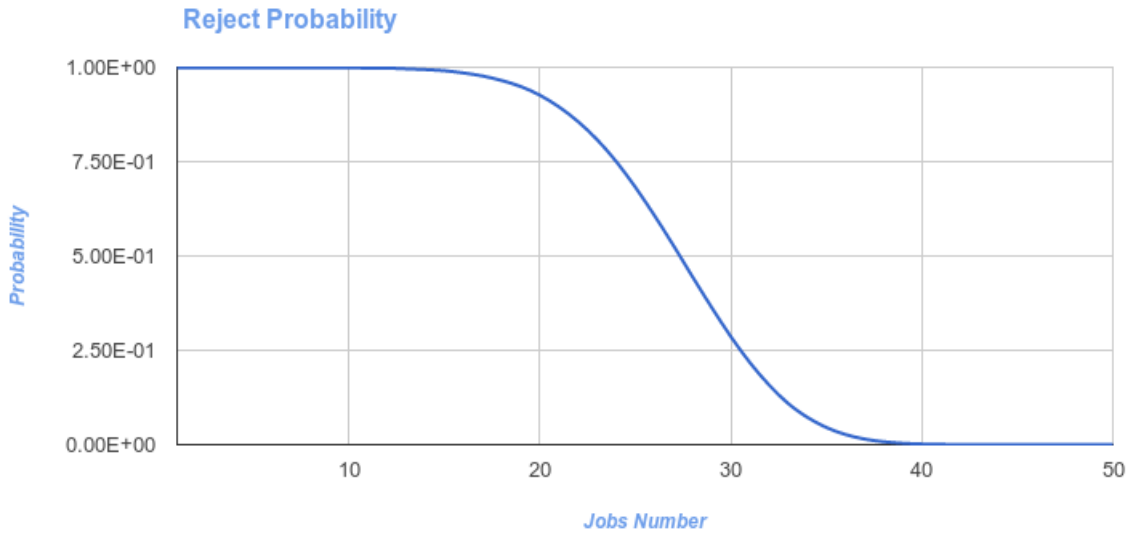

Figura 4.4

Si noti come la probabilità per un job di essere rifiutato scenda rapidamente all'aumentare della soglia da un valore di circa 15 in poi, arrivando ad essere praticamente nulla in prossimità di una soglia pari a 40. Per quanto osservato in funzione del fattore di utilizzazione del FE Server quando nel sistema sono presenti 50 jobs, c'è da aspettarsi che l'introduzione della soglia di rifiuto migliori il tempo di risposta complessivo del sistema, che allo stato si aggira intorno ai 300 secondi (5 minuti).

Come illustreremo nella sezione relativa ai risultati della soluzione del secondo modello, l'effetto dell'introduzione della soglia sarà più notevole quando il sistema è popolato da un numero di job elevato ma inferiore a 50 (valore per il quale il sistema è totalmente congestionato). Empiricamente mostreremo tale valore attestarsi in un range che va da 39 a 46 jobs presenti nel sistema.

4.1.6 Requisito della QoS riguardo alla Probabilità di Rejection

Nelle specifiche progettuali, è espresso un valore massimo per la probabilità di rejection:

$$p_{ci,ci-rej} \leq 0.22$$

Per rispettare quanto richiesto dalla QoS dell'impianto, è possibile assumere solo quei valori di soglia S che impongono una probabilità di rifiuto non superiore a 0.22, ovvero esistono in funzione di N dei valori di S sotto ai quali non è possibile andare senza violare il requisito suddetto. Tali valori sono illustrati nella figura seguente per N=50:

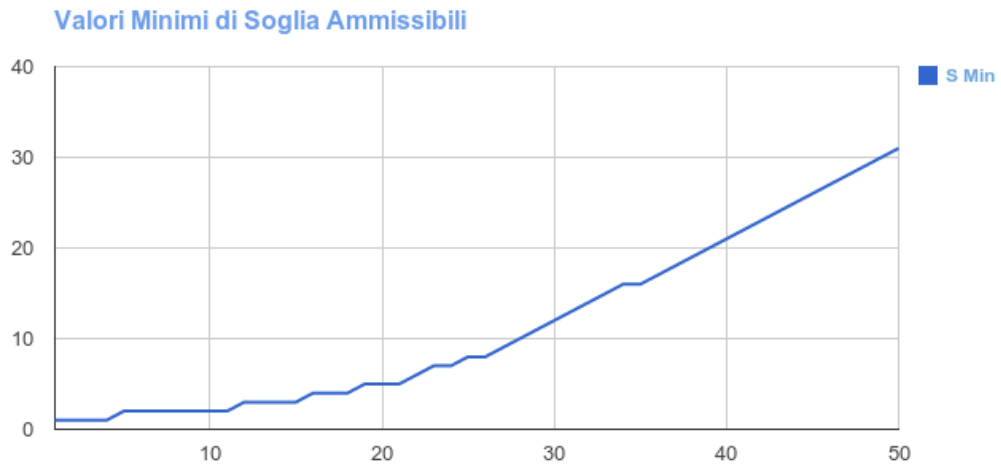

Figura 4.5

N	S min	N	S min	N	S min	N	S min	N	S min
1	1	11	2	21	5	31	13	41	22
2	1	12	3	22	6	32	14	42	23
3	1	13	3	23	7	33	15	43	24
4	1	14	3	24	7	34	16	44	25
5	2	15	3	25	8	35	16	45	26
6	2	16	4	26	8	36	17	46	27
7	2	17	4	27	9	37	18	47	28
8	2	18	4	28	10	38	19	48	29
9	2	19	5	29	11	39	20	49	30
10	2	20	5	30	12	40	21	50	31

Tabella 4.2

4.2 Soluzione del secondo impianto

Come anticipato nel paragrafo 3.2.2, per parametrizzare il secondo modello descritto nelle specifiche del progetto, abbiamo ottenuto i valori per le soglie S1 e S2 che minimizzassero il tempo di risposta del sistema sperimentato dai job del Client 1 sviluppando un nuovo modello, che da ora in avanti chiameremo Modello di Supporto SM (Support Model).

Nella presente sezione mostreremo dapprima i risultati del meccanismo di tuning dell'impianto così implementato, spiegando gli effetti della variazione del tempo di risposta del sistema per Client 1 al variare del numero di job del sistema e del valore delle soglie S1 e S2, per poi illustrare la soluzione del modello originale parametrizzato con i valori ottenuti.

4.2.1 Minimizzazione del Tempo di Risposta

Per ottenere il tempo di risposta sperimentato da un job proveniente per la prima volta da Client 1, che modella il primo tentativo da parte di un utente di instaurare una sessione con il sistema server, abbiamo risolto il modello di supporto SM descritto nel paragrafo 3.2.2.

Per la giustificazione della necessità e delle scelte modellistiche riguardanti detto modello, rimandiamo il lettore al paragrafo 3.2.3.

Dalla soluzione tramite MVA del modello SM al variare dei valori delle soglie S1 e S2 e del numero N di jobs nel sistema, abbiamo ottenuto i seguenti valori minimi per il tempo di instaurazione e durata di una sessione:

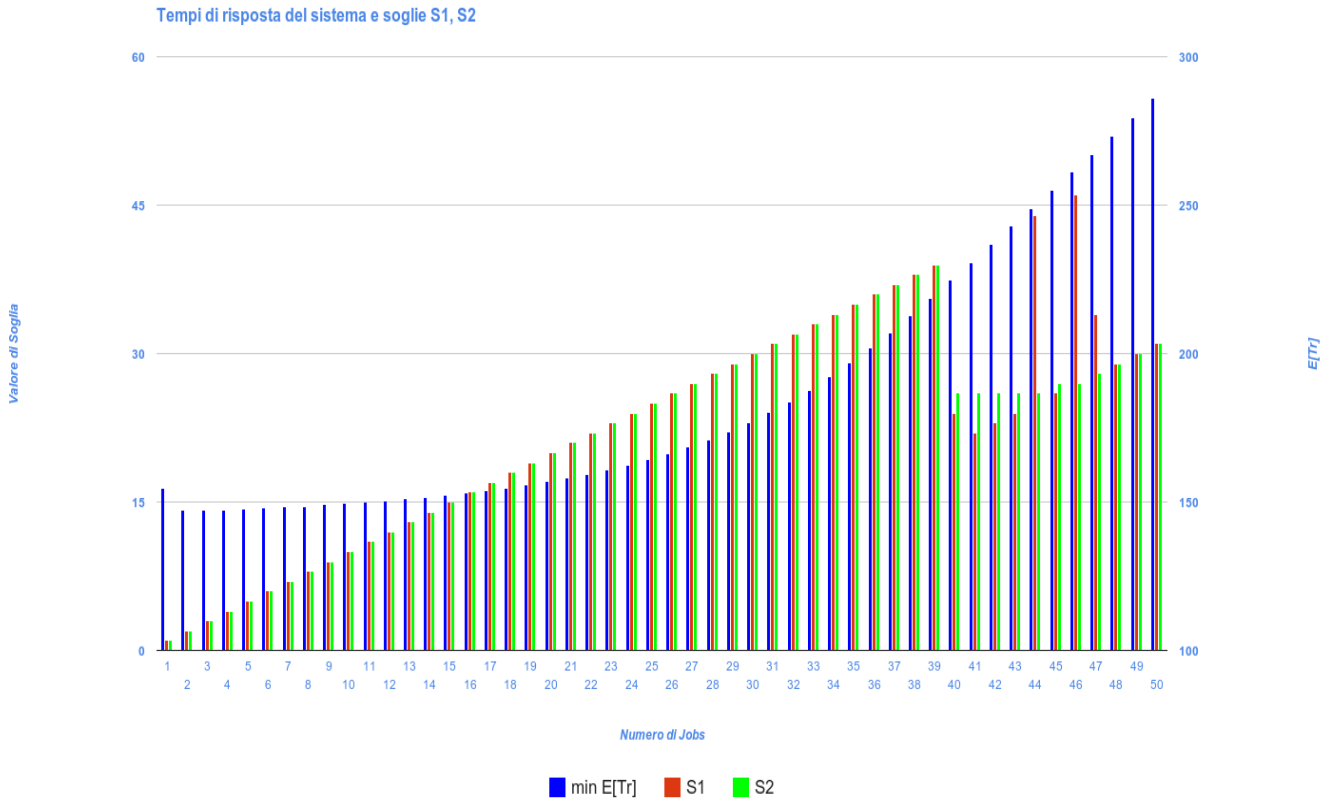

Figura 4.6

N	S1	S2	$E[t_r]$	N	S1	S2	$E[t_r]$
1	1	1	154.723214	26	26	26	166.256702
2	2	2	147.401095	27	27	27	168.49588
3	3	3	147.101913	28	28	28	170.979298
4	4	4	147.307182	29	29	29	173.728697
5	5	5	147.59907	30	30	30	176.765022
6	6	6	147.919207	31	31	31	180.107442
7	7	7	148.266382	32	32	32	183.772188
8	8	8	148.64034	33	33	33	187.771314
9	9	9	149.044396	34	34	34	192.111447
10	10	10	149.481419	35	35	35	196.792692
11	11	11	149.955295	36	36	36	201.807837
12	12	12	150.470129	37	37	37	207.142008
13	13	13	151.030767	38	38	38	212.77289
14	14	14	151.642591	39	39	39	218.671577
15	15	15	152.311783	40	24	26	224.712858
16	16	16	153.045307	41	22	26	230.759917
17	17	17	153.851083	42	23	26	236.812695
18	18	18	154.73805	43	24	26	242.854262
19	19	19	155.716319	44	44	26	248.927462
20	20	20	156.797253	45	26	27	254.96304
21	21	21	157.993599	46	46	27	261.120431
22	22	22	159.319561	47	34	28	267.07502
23	23	23	160.790869	48	29	29	273.174823
24	24	24	162.424782	49	30	30	279.448894
25	25	25	164.24004	50	31	31	285.880294

Tabella 4.2

Osservando l'istogramma in figura 4.6, notiamo un andamento delle soglie che può essere suddiviso in funzione dei valori di N. Analizziamo di seguito l'andamento dei valori delle soglie da 1 a 39 e da 40 a 50, dando poi giustificazione della nostra interpretazione.

Valori di N da 1 a 39

Quando il numero di utenti nel sistema è all'interno del range [1,39], l'introduzione di una soglia, sia per i job provenienti da **Client 1** che per quelli provenienti da **Client 2**, risulta essere sconveniente, in quanto il ritardo introdotto a causa della rejection dei job risulta degradare il tempo di risposta più di quanto non faccia l'introdurre nei server nuovi jobs.

In figura 4.6, quando N è minore o uguale a 39, il valore delle due soglie è sempre pari al numero dei jobs complessivamente presenti nel sistema. Questo equivale di fatto ad aver disattivato il meccanismo di rejection: tutti i job entrano nei server senza essere rifiutati.

Valori di N da 40 a 50

In figura 4.6 si può notare come, in presenza di 40 o più jobs nel sistema, i due meccanismi di rejection vengono attivati con intensità variabile. Le due soglie decrescono all'aumentare del numero di jobs da 40 a 41, per poi aumentare gradualmente. Tale aumento però, per $N \geq 38$, è dovuto all'assunzione dei valori minimi che una soglia può assumere rispettando il requisito sulla QoS (esposti nella Tabella 4.1):

$$p_{ci,ci-rej} \leq 0.22$$

Oltre a ciò, relativamente alla soglia S1, si notano dei picchi in corrispondenza dei valori di N= 44 e N=46, nei quali la soglia viene nuovamente disattivata.

Si desume quindi che al variare di N non è scontato determinare se sia maggiore il ritardo dovuto all'introduzione della soglia S1, con conseguente riesecuzione presso il client 1, o piuttosto il degrado prestazionale del sistema dovuto all'introduzione di un numero di jobs maggiore.

Dedichiamo particolare attenzione a quest'ultimo risultato dandone un'analisi più approfondita nel paragrafo seguente.

4.2.2 Analisi della Variazione del Tempo di Risposta del Sistema in funzione di S1-S2

In questa sezione diamo giustificazione di quanto asserito nel paragrafo precedente, ovvero dell'esistenza di due effetti contrapposti dovuti all'attivazione delle soglie di rifiuto:

1. miglioramento del tempo di risposta del sistema dovuto ad un minor carico
2. reimposizione del tempo di think presso il Client che ha subito la rejection

Quando il numero di utenti presenti nel sistema è inferiore a 39, abbiamo visto come di fatto sia conveniente disattivare il meccanismo di rejection, ovvero attribuire alle soglie S1 e S2 il valore massimo per permettere ad ogni job di accedere al servizio.

Nella figura seguente vediamo come valori alti delle soglie influenzino positivamente il tempo di risposta globale "visto" dal **Client 1:**

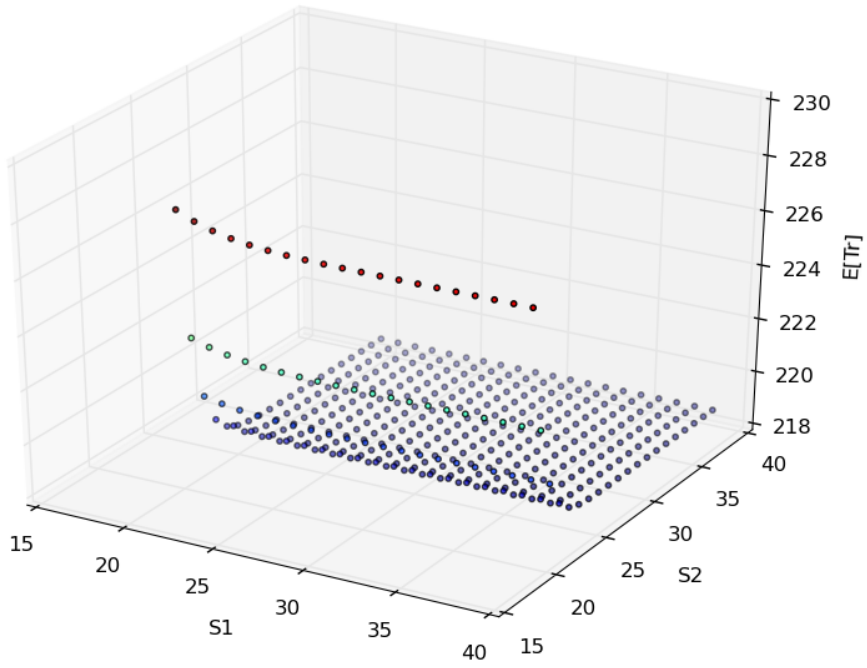

Figura 4.7 - Soglie S1, S2 per N=39

Nel grafico si nota subito come il tempo di risposta minimo è da cercarsi fra i valori massimi di S1,S2 (ovvero 39).

Nella figura seguente, dove invece consideriamo il caso per N=40 , è appena percettibile in quale modo il contributo delle soglie ottimizzi il tempo di risposta: infatti non si distingue bene a occhio nudo la “deformazione” della superficie vicina al piano **xy,** che comporta l’adozione di soglie apparentemente in contraddizione con quanto osservato precedentemente.

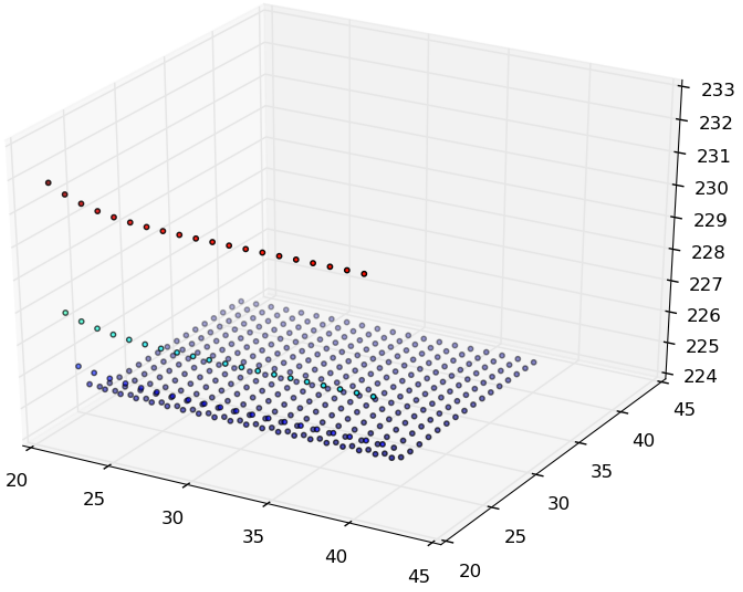

Figura 4.8 - Soglie S1, S2 per N=40

Nelle figure che seguono la “deformazione” della superficie descritta dai punti del grafico diventa decisamente più evidente. In particolare si nota l’effetto contrapposto dei due contributi dovuti all’inasprirsi del valore delle soglie, descritti all’inizio di questo paragrafo.

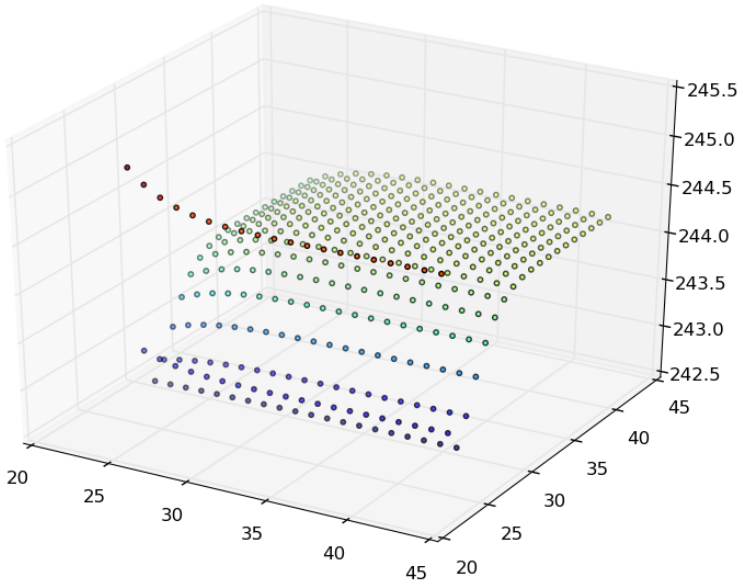

Figura 4.9 - Soglie S1, S2 per N=43

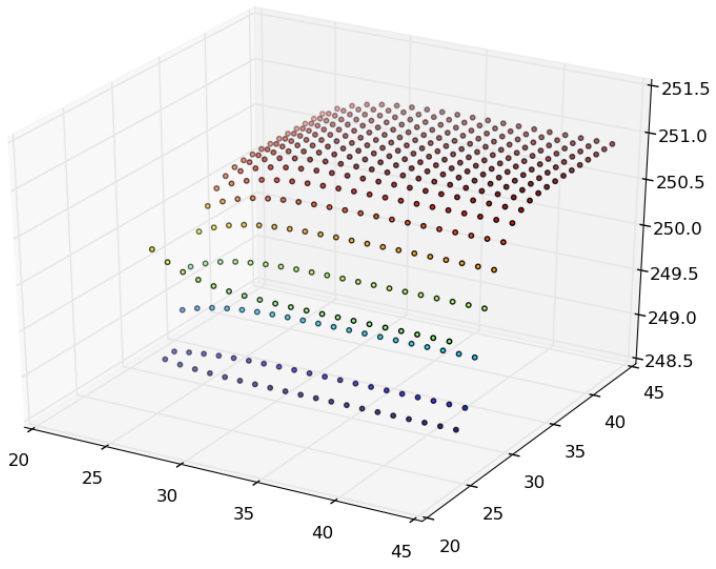

Figura 4.10 - Soglie S1, S2 per N=44

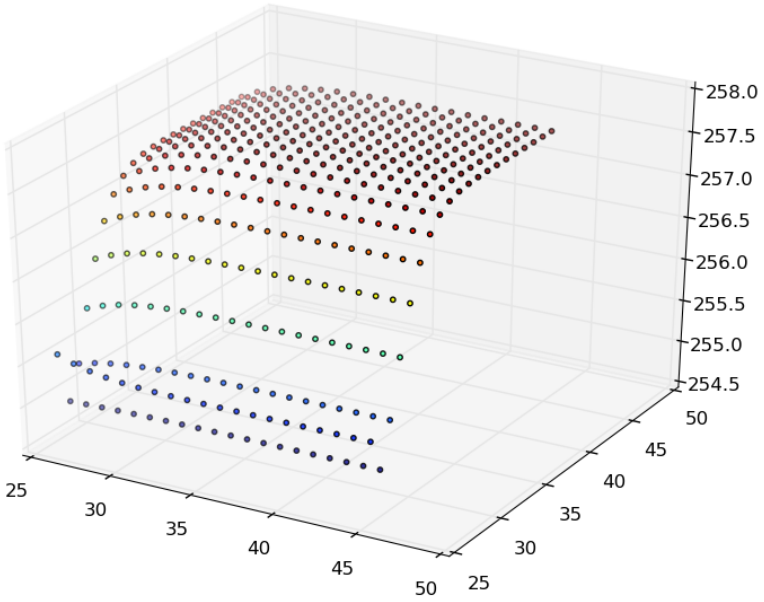

Figura 4.11 - Soglie S1, S2 per N=45

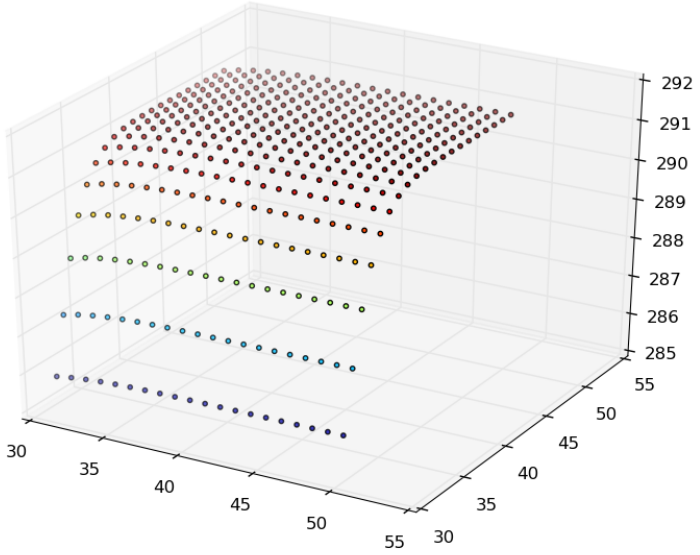

Figura 4.12 - Soglie S1, S2 per N=50

4.2.3 Parametrizzazione e Soluzione del Secondo Modello

Il secondo modello di impianto, esposto nelle specifiche del presente progetto, è stato risolto al variare del numero di utenti N nel sistema, dopo essere stato parametrizzato con gli opportuni valori di soglia S1 e S2, tramite l'algoritmo in forma prodotto MVA.

Per la scelta dei parametri S1 e S2, rimandiamo il lettore ai paragrafi precedenti (4.2.1 e 4.2.2).

Di seguito illustriamo gli indici ottenuti dalla soluzione del modello.

Tempi di Risposta Locali

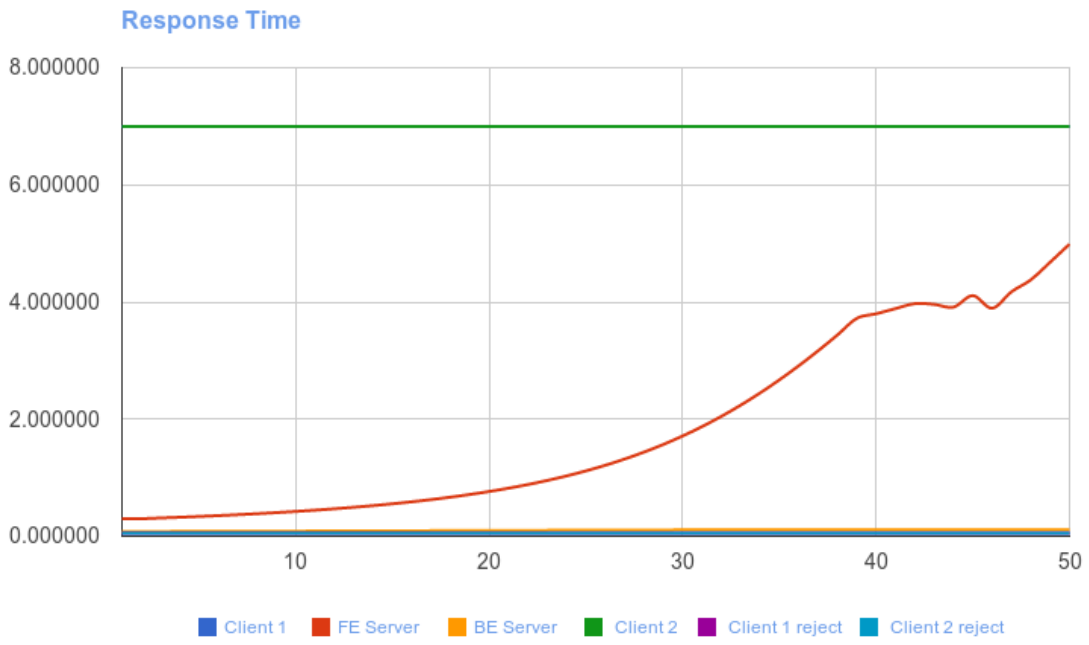

Figura 4.13

Poiché il tempo di risposta di un Infinite Server è pari al solo tempo di servizio, ci soffermiamo ad analizzare i risultati relativi al **FE Server** ed al **BE Server**.

Dal grafico in Figura 4.13, notiamo come il tempo di risposta del **FE Server** evidenzi come tale centro costituisca il collo di bottiglia dell'impianto, mentre il BE Server sia sostanzialmente scarico anche in presenza del massimo valore di jobs considerato.
In particolare, l'andamento del tempo di risposta del FE Server incrementa considerevolmente quando nel sistema è presente un numero di jobs superiore a 30, mentre il crescere della derivata della sua curva è interrotto soltanto quando si iniziano a manipolare i valori delle soglie.

In altre parole, quando viene attivato il meccanismo di reject per i job che accedono al servizio diminuisce la velocità con cui cresce il tempo di risposta del FE Server.

Quando invece nel sistema sono presenti almeno 46 jobs, si nota un nuovo incremento della derivata, che si attesta su un valore praticamente costante. Ciò significa che il sistema è sostanzialmente in saturazione, e il tempo di risposta cresce praticamente in modo lineare rispetto al numero di jobs presenti nel sistema.

Throughputs Locali

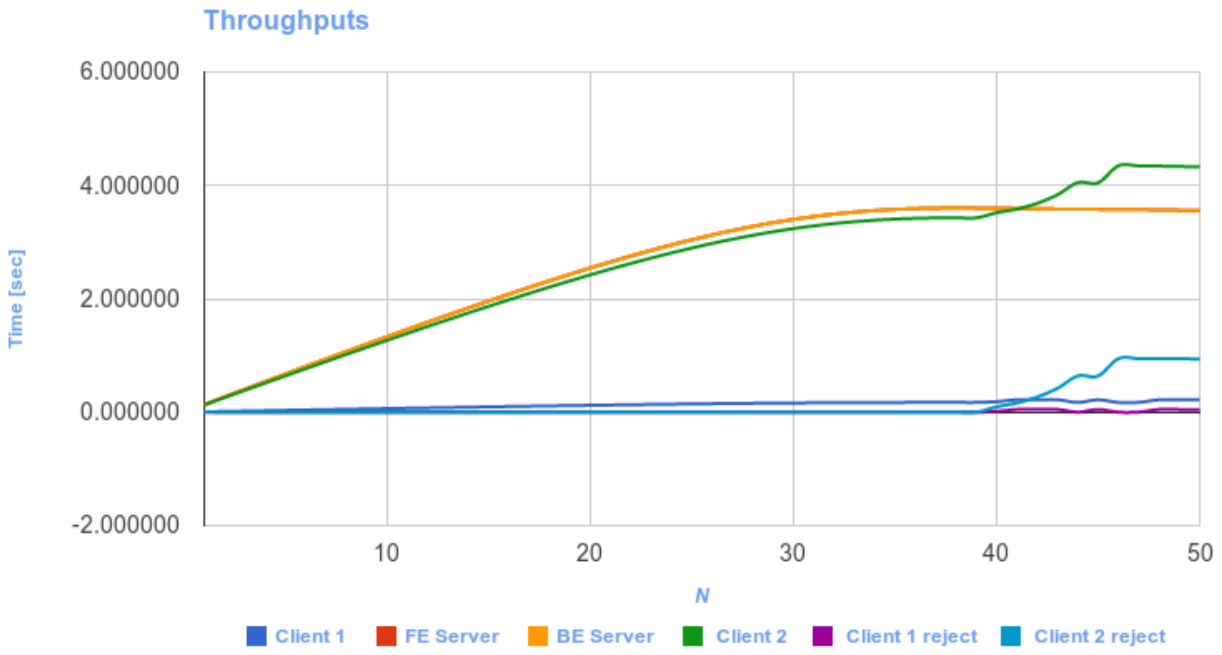

Figura 4.14

Dal modello dell'impianto, è facile convincersi di quanto evidenziato in figura 4.14 riguardo i throughput di FE Server e BE Server: essi sono infatti coincidenti, trovandosi i due centri in serie sullo stesso ramo del modello.

Si noti l'andamento del throughput dei centri **Client 2** e **Client 2rej**, che sono fra loro assimilabili a meno di un fattore motiplicativo, e che subiscono visibili fluttuazioni in corrispondenza dell'attivazione del meccanismo di rejection, ovvero a partire da quando nel sistema sono presenti almento 40 jobs.

Fluttuazioni meno visibili ma comunque degne di nota sono presenti anche presso **Client 1** e **Client 1 rej**, e si giustificano in modo del tutto analogo a quanto detto per **Client 2** e **Client 2 rej**.

Popolazioni Medie Locali

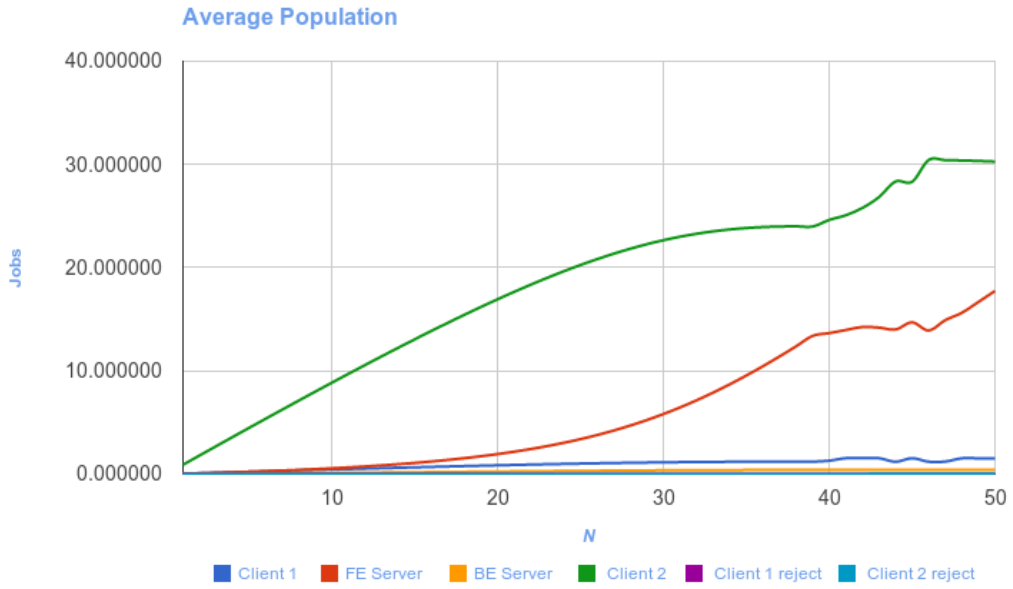

Figura 4.15

Come si era già notato nella soluzione del primo modello, i jobs dell'impianto tendono ad accumularsi in coda al **FE Server** oppure in attesa del tempo di think presso **Client 2.**

Anche in questo caso, l'attivazione del meccanismo di rejection (sempre in corrispondenza di N=40), si ripercuote sull'andamento del grafico in figura 4.15: l'andamento della media della popolazione presso i centri subisce distinguibili fluttuazioni.

Notiamo in particolare come la popolazione presso **FE Server** incrementi con derivata praticamente costante per valori di N $\geq$46.

Indici Globali

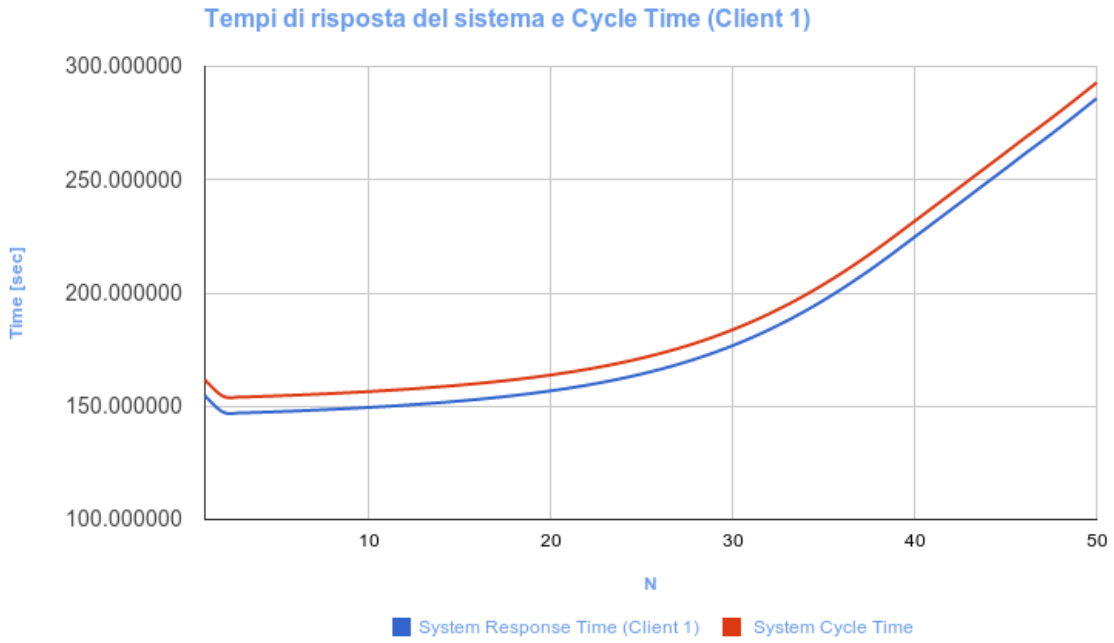

Figura 4.17

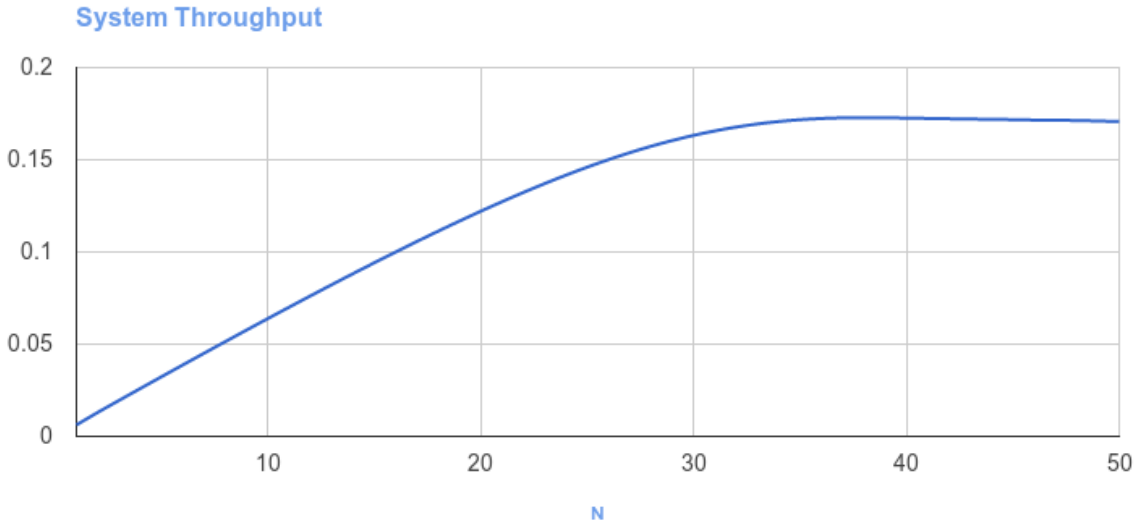

Figura 4.18

La figura 4.18 illustra l'andamento del tempo di risposta del sistema e del Cycle Time rispetto al **Client 1**. Notiamo come il carico subito dal sistema, quando i jobs al suo interno sono più di 30, è tale da imporre una crescita sempre più sostenuta del tempo di risposta.

Il grafico sul throughput mostra come, a seguito dell'attivazione delle soglie, il flusso di jobs all'interno del sistema viene in parte "assorbito" all'interno del meccanismo di rejection.
Di fatto, vengono introdotti nuovi cicli che sottraggono jobs dal circuito principale.

APPENDICE A : Listing del codice

Appendice A.1 - main.py

```
'''
Created on 20/dic/2013

@author: Emanuele Paracone
@contact: emanuele.paracone@gmail.com
@author: Serena Mastrogiacomo
@contact: serena.mastrogiacomo@gmail.com

'''


import cPickle as pickle
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D



from gordonNewell.gordonNewell import GordonNewell
from mva.mva import MVA

# the minimum threasholds array, obtained from Gordon Newell execution
minSs = [1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 7, 7, 8, 8, 9, 10,\
        11, 12, 13, 14, 15, 16, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]



# Run Gordon Newell
def gnRun():
    print 'Risoluzione del primo modello tramite Gordon Newell'
    s = raw_input('\t1. esecuzione standard\n\t2.stampa vettore probabilità in funzione del numero di job N\n[1,2]:')
    if s[0]=='1':
        s = raw_input('inserire N:')
        n = int(s)
        gn = GordonNewell(n)
        gn.solve()
        gn.printS()
        gn.printIndexes()
    else:
        probList = []
        for n in range(1,51):
            gn = GordonNewell(n)
            gn.solve(False)
            probList.append(gn.getSProbsList())
        print 'serializing...'
        out_pkl = open('probs.pkl','wb')
        pickle.dump(probList,out_pkl)
        print 'done!'


# run a single execution of MVA
def MVARun():
    print 'Risoluzione del secondo modello tramite MVA'
    print 'ricerca delle soglie S1, S2 ottimali rispetto al tempo di risposta dell\'impianto visto dal centro client 1...'
```

```
    sMin = 31
    m = 6
    s = raw_input('inserire n:')
    n = int(s)

    s1 = 100
    s2 = 100
    eTrMin = 10000.0
    leastEtrMva = None
    mvas = []
    for i in range(sMin,n+1):
        for j in range(sMin,n+1):
            mvas.append(MVA(1,m,n,i,j))
            mvas[-1].solve()
            mvas[-1].localIndexesCompute()
            mvas[-1].globalIndexesCompute()
            eTr = mvas[-1].systemResponseTimeClient1()
            if eTr < eTrMin:
                eTrMin = eTr
                leastEtrMva = mvas[-1]
    leastEtrMva.printIndexes()
    leastEtrMva.printFirstIndexes()

# look for the threasholds which minimizes the system response time for Client1 jobs through multiple runs of
MVA
# and write the files of the solved original model parametrized with found values
def MVAThreasholdSearchRun( outputFile = None):
    print 'Risoluzione del secondo modello tramite MVA'
    print 'esecuzione dei possibili modelli al variare di N'



    m = 6
    nMax = 50
    responseTimesFile = open('response_times.csv','w')
    throughputsFile = open('throughputs.csv','w')
    utilizationsFile = open('utilizations.csv','w')
    populationsFile = open('populations.csv','w')
    pointsLists = []

    responseTimesFile.write('N,Client 1,FE Server,BE Server, Client 2, Client 1 reject, Client 2 reject,\n')
    throughputsFile.write('N,Client 1,FE Server,BE Server, Client 2, Client 1 reject, Client 2 reject,\n')
    utilizationsFile.write('N,Client 1,FE Server,BE Server, Client 2, Client 1 reject, Client 2 reject,\n')
    populationsFile.write('N,Client 1,FE Server,BE Server, Client 2, Client 1 reject, Client 2 reject,\n')


    pointsLists.append([])
    for n in range(1,nMax+1):
        pointsLists.append([])
        pointsLists[n].append([])
        pointsLists[n].append([])
        pointsLists[n].append([])

        s1 = 100
        s2 = 100
        eTrMin = 10000.0

        mvas = []
        sMin = minSs[n-1]
        for i in range(sMin,n+1):
```

```
            mvas.append([])
            for j in range(sMin,n+1):
                mvas[i-sMin].append(MVA(2,m,n,i,j))
                mvas[i-sMin][-1].solve()
                mvas[i-sMin][-1].localIndexesCalculate()
                mvas[i-sMin][-1].globalIndexesCalculate()
                eTr = mvas[i-sMin][-1].systemResponseTimeClient1()
                pointsLists[n][0].append(i)
                pointsLists[n][1].append(j)
                pointsLists[n][2].append(eTr)

                if eTr < eTrMin:
                    eTrMin = eTr
                    s1 = mvas[i-sMin][-1].getS1()
                    s2 = mvas[i-sMin][-1].getS2()

        print '\tN=%d  min eTr:%f\ts1:%d\ts2:%d'%(n,eTrMin,s1,s2)
        if outputFile:
            outputFile.write('%d,%f,%d,%d,\n'%(n,eTrMin,s1,s2))
        # solve the original model with S1, S2 found
        originalMva = MVA(1,m,n,s1,s2)
        responseTimesFile.write('%d,%s\n'%(n,originalMva.responseTimesToString()))
        throughputsFile.write('%d,%s\n'%(n,originalMva.throughputAvgsToString()))
        utilizationsFile.write('%d,%s\n'%(n,originalMva.utilizationAvgsToString()))
        populationsFile.write('%d,%s\n'%(n,originalMva.populationAvgsToString()))

    #closing files
    responseTimesFile.close()
    throughputsFile.close()
    utilizationsFile.close()
    populationsFile.close()
    s = raw_input('stampare i grafici del tempo di risposta del sistema subito dal client1?\n[s,n]:')
    if s[0]=='s':
        plotAllPoints(pointsLists)



# the plotting points function - plot a 3d graph using matplot library with mpl toolkit extension
def plotAllPoints(pointsLists):
    print 'plotting points...'
    for n in range (1,len(pointsLists)):
        fig = plt.figure(figsize=(10,7 ), dpi=100)
        ax3D = fig.add_subplot(111, projection='3d')
        x = pointsLists[n][0]
        y = pointsLists[n][1]
        z = pointsLists[n][2]
        ax3D.scatter(x, y, z, s=10, c=z, marker='o')
        plt.savefig('threasholds_N%d.png'%(n))
        plt.clf()
    print 'done!'

# the global main function!
if __name__=="__main__":
    print '\n\n================================================='
    print '=\t Modelli di Prestazione di \t\t='
    print '=\t      Sistemi e Reti\t\t\t='
    print '=\t\t2012-2013\t\t\t='
    print '================================================='

    while True:
        s1 = raw_input('\n\nModello da risolvere:\n\t1. modello 1 (algoritmo Gordon-Newell)\
```

```
    \n\t2. modello 2 (algoritmo MVA semplice, parametri custom)\n\t3. modello 2 (algoritmo MVA, ricerca soglie
minime al variare del numero di utenti)\n\n[1,2,3]:')
    if s1[0]=='1':
        gnRun()

    elif s1[0]=='2':
        MVARun()

    elif s1[0]=='3':
        outputFile = open('thresholds.csv','w')
        MVAThreasholdSearchRun(outputFile)
        outputFile.close()

    elif s1=='exit' or s1=='quit' :
        print 'bye!'
        break
    else:
        print 'inserire 1, 2, 3 o exit'
```

Appendice A.2 - gordonNewell.py

```
#! /usr/bin/python
#coding: utf-8
'''
Created on 20/dic/2013

@author: Emanuele Paracone
@contact: emanuele.paracone@gmail.com
@author: Serena Mastrogiacomo
@contact: serena.mastrogiacomo@gmail.com

'''

import datetime
from math import factorial

class GordonNewell():

    # infinite server declaration between centers
    IS = [True,False,False,True]
    # relative utilization factors
    xs = [1.0, 0.892857108, 0.238095229, 19.83333256 ]
    # the average service times for centers
    ts = [7.0, 0.3, 0.08, 7.0]
    # declaring global variables
    responseTimeAvgs = None
    cycleTimeAvg = None
    throughput = None
    s = None
    sProb = None
    #initializing the lower bound for the probability for a job to be accepted
    sProbBound = 0.78

    # the constructor
    def __init__(self, N=50,M=4):
```

```
        # initializing the jobs number
        self.N = N
        # declaring centers Number
        self.M = M
        # the relative throughput ratio between centers
        self.vs=[]
        # the service tax for centers
        self.mu = []
        # the productiories for relative utilization factors array
        self.produttorie = []
        # the normalization factor G(N)
        self.normalizationCostant = 0
        # the state probabilities array
        self.stateProbabilities = []
        # initializing the normalization factor sum
        self.sum=0
        # the centers throughputs array
        self.lambdas = []
        # the utilization factors array
        self.rhos = []
        # the average local populations array
        self.nLocalAvg = []
        # the average service times array
        self.executionTimeAvgs = []
        self.array = []
        # the states array
        self.states = []
        # the threasholds probabilities array
        self.sProbs = []
        # the marginal probabilities array
        self.marginalProbabilities = []
        # initializing the average system response times for centers
        self.responseTimeAvg = 0
        # initializing the service taxes
        for i in range(M):
            self.array.append(0)
            self.mu.append(1.0/self.ts[i])


    # building the states space
    def generaStati(self, index, jobNum):
        for i in reversed(range(jobNum+1)):
            tmpJobN = jobNum -i
            self.array[index]=i
            if (tmpJobN > 0) and ( (index+1) < self.M) :
                self.generaStati(index+1, tmpJobN)
            else:
                for j in range (index+1, len(self.array)):
                    self.array[j]=0
                if tmpJobN == 0:
                    self.addState();

    # add a new state to the states array
    def addState(self):
        tmpArray = []
        for i in range (self.M ):
            tmpArray.append(self.array[i])
        self.states.append(tmpArray)

    # print states
    def printStates(self):
```

```
        for i in range(len(self.states)):
            print '%d. %s'%(i,str(self.states[i]))

    # initialize the productories array
    def produttorieInit(self):
        for i in range(len( self.states)):
            tmpProd = 1
            for j in range (self.M):
                tmpExp = self.xs[j]**self.states[i][j]
                if self.IS[j]:
                    tmpProd*= float(tmpExp)/factorial(self.states[i][j])
                else:
                    tmpProd*= tmpExp


            self.produttorie.append(tmpProd)



    # compute the normalization constant G(N)
    def normalizationCostantCompute(self):
        for produttoria in self.produttorie:
            self.normalizationCostant+=produttoria

    # compute all the state probabilities
    def stateProbabilitiesCompute(self):
        self.sum =0.0
        for produttoria in self.produttorie:
            self.stateProbabilities.append( float(produttoria) / self.normalizationCostant )
            self.sum+=self.stateProbabilities[-1]
            if self.stateProbabilities[-1] > 1:
                print 'ops!'

    # write the csv file of state probabilities
    def makeCsv(self):
        path='state_probabilities.csv'
        f=open(path,"w")
        f.write('state, probability,\n')
        for i in range(len(self.states)):
            f.write('%s,%s,\n'%(str(self.states[i]).replace(',',';'), str(self.stateProbabilities[i])))
        f.close()


    #---------------------------------------- local indexes---------------------------------


    # compute the centers utilization factors
    def rhosCompute(self):
        for j in range (self.M):
            rhoJ = 0
            for i in range (len(self.states)):
                if (self.states[i][j]>0):
                    rhoJ+=self.stateProbabilities[i]
            self.rhos.append(rhoJ)

    # compute the throughputs for centers
    def lambdasCompute(self):
        for i in range(self.M):
            self.lambdas.append( self.rhos[i]* self.mu[i])

    # compute the average populations fro centers
```

```
def nLocalAvgCompute(self):
    for j in range(self.M):
        self.nLocalAvg.append(0)
        for i in range(len(self.states)):
            self.nLocalAvg[-1] += self.states[i][j]*self.stateProbabilities[i]

# compute the average execution time for centers
def executionTimeLocalAvg(self):
    for j in range (self.M):
        if self.IS[j]:
            self.executionTimeAvgs.append( self.ts[j] )
        else:
            self.executionTimeAvgs.append( float(self.nLocalAvg[j])/self.lambdas[j] )

# compute the centers average response times
def responseTimeLocalAvg(self):
    self.vsCompute()
    self.responseTimeAvgCompute()

# compute the marginal probabilities
def marginalProbabilitiesCompute(self):
    for j in range (self.M):
        self.marginalProbabilities.append([])
        for i in range (self.N+1):
            self.marginalProbabilities[-1].append(0)
    for i in range (len (self.states)):
        for j in range(self.M):
            tmp=self.states[i][j]
            if tmp > 0:
                self.marginalProbabilities[j][tmp]+= self.stateProbabilities[i]

# compute the relative throughputs
def vsCompute(self):
    for i in range(self.M):
        self.vs.append([])
        for j in range (self.M):
            self.vs[i].append(float(self.lambdas[i])/self.lambdas[j])

# compute the system response time averages
def responseTimeAvgCompute(self):
    self.responseTimeAvgs = []
    for j in range (self.M):
        self.responseTimeAvgs.append(0)
        for i in range (self.M):
            if i == j:
                continue
            self.responseTimeAvgs[j]+= self.vs[i][j]*self.executionTimeAvgs[i]

# ------------------------------------------------ global indexes ------------------------------------

# compute the system response time for Client 1
def responseTimeAvgGlobalCompute(self):
    # we use the first center as reference
    for i in range (1,self.M):
        self.responseTimeAvg += self.vs[i][1] * self.responseTimeAvgs[i]

# compute the system cycle time for Client 1
def cycleTimeAvgGlobalCompute(self):
    self.cycleTimeAvg = self.responseTimeAvg + self.responseTimeAvgs[0]

# compute the system throughput for Client 1
```

```
def throughputCompute(self):
    self.throughput = float(self.N)/ self.cycleTimeAvg

# find the lower value of the threashold S which is higher than the lower bound
def findLowerS(self):
    for i in range (self.N+1):
        self.sProbs.append(0)
    for i in range(len(self.states)):
        tmp = self.states[i][1]+self.states[i][2]
        self.sProbs[tmp]+=self.stateProbabilities[i]
    tmpSum =0.0
    for i in range(self.N+1):
        if tmpSum >self.sProbBound:
            self.s=i
            self.sProb = 1.0-tmpSum
            break
        tmpSum += self.sProbs[i]

# returns the minimum value for a reject threashold S
def getS(self):
    return self.s

# returns the maximum threashold probability
def getSProb(self):
    return self.sProb

# prints all the threasholds probabilities
def printS(self):
    tmpSum=0.0
    print 'soglie:'
    for i in range(self.N+1):
        print '\t%d. %0.20f'%(i,1.0-tmpSum)
        tmpSum+=self.sProbs[i]
    print 'La soglia s è %d'%(self.s)
    print 'costante norm:%0.20f'%(self.normalizationCostant)

# compute all the local indexes
def localIndexesCompute(self):
    self.rhosCompute()
    self.lambdasCompute()
    self.nLocalAvgCompute()
    self.executionTimeLocalAvg()
    self.responseTimeLocalAvg()

# compute all the global indexes
def globalIndexesCompute(self):
    self.responseTimeAvgGlobalCompute()
    self.cycleTimeAvgGlobalCompute()
    self.throughputCompute()

# print all the threasholds probabilities
def printSProb(self):
    tmpSum = 0.0
    probs = []
    for i in range(self.N+1):
        probs.append(1.0-tmpSum)
        tmpSum+=self.sProbs[i]
    print probs

# returns the array of threasholds prbabilities ( which will be useful for parametrizing mva)
def getSProbsList(self):
```

```
        tmpSum = 0.0
        probs = []
        for i in range(self.N+1):
            probs.append(1.0-tmpSum)
            tmpSum+=self.sProbs[i]
        return probs

    # print all indexes, both local and global
    def printIndexes(self):
        print '================================================'
        print '=\t\tLOCAL INDEXES\t\t='
        print '================================================'
        for i in range (self.M):
            print '\ncenter %d:\n\tmu:\t\t\t%0.20f\n\tlambda:\t\t\t%0.20f\n\tutilizationFactor:\t%0.20f\n\tEn:\t\
            \t%0.20f\n\tEtr:\t\t\t%0.20f\n\tEt:\t\t\t%0.20f\n\trho:\t\t\t%0.20f\n'%(i+1,self.mu[i],self.lambdas[i],self.rhos[i],\
                                        self.nLocalAvg[i],self.responseTimeAvgs[i],self.executionTimeAvgs[i],self.rhos[i])
        print '================================================'
        print '=\t\tGLOBAL INDEXES\t\t='
        print '================================================'
        print 'System Response Time:%0.20f\nSystem Cycle Time:%0.20f\nSystem Troughput:%0.20f'%\
        (self.responseTimeAvg,self.cycleTimeAvg,self.throughput)

    # the Gordon Newell algorithm solver
    def solve(self, verbose=True):
        if verbose:
            print 'generate states...'
            self.generaStati(0, self.N)
            print 'done.'
            print 'Compute state probailities...'
            self.produttorieInit()
            self.normalizationCostantCompute()
            self.stateProbabilitiesCompute()
            print 'done.'
            print 'find lower bound for S (useful for next exercise)...'
            self.findLowerS()
            print 'done'
            print 'calculating local indexes...'
            self.localIndexesCompute()
            print 'done.'
            print 'calculating global indexes...'
            self.globalIndexesCompute()
            print 'done'
        else:
            self.generaStati(0, self.N)
            self.produttorieInit()
            self.normalizationCostantCompute()
            self.stateProbabilitiesCompute()
            self.findLowerS()
            self.localIndexesCompute()
            self.globalIndexesCompute()

# the main function for the class
if __name__=="__main__":
    print 'Gordon Newell !!!'

    m = 4
    while True:
        print '----------------------------------------'
        s = raw_input('\t1. Gordon Newell\n\t2. Print thresholds array\n\t[1,2]:')
        if s[0] == '1':
            s = raw_input('inserire n:')
```

```
            n = int(s)
            gn = GordonNewell(n,m)
            gn.solve()
            gn.printS()
            gn.printIndexes()
            print 'done!'
        elif s[0] =='2':
            #the array of minimum thresholds
            ss = []
            for n in range(1,51):
                print 'solving Gordon Newell algorithm for n:%d...'%(n),
                gn = GordonNewell(n,m)
                gn.solve(False)
                ss.append(gn.getS())
                print ' done!'
            print 'ss: %s'%(str(ss))
        else:
            print 'bye!'
            break
```

Appendice A.3 -mva.py

```
#! /usr/bin/python
#coding: utf-8

'''
Created on 20/dic/2013

@author: Emanuele Paracone
@contact: emanuele.paracone@gmail.com
@author: Serena Mastrogiacomo
@contact: serena.mastrogiacomo@gmail.com

'''

import datetime
import cPickle as pickle

class MVA():

    # customers number
    N = None
    # centers number in the system
    M = None

    # thresholds
    s1 = None
    s2 = None

    # thresholds probabilities
    ps1 = None
    ps2 = None
```

```
    # relative throughput matrix
    ys = None
    # relative throughput ratio matrix
    vs = None
    # service time for jobs for each center
    eTs = None
    # infinite server declaration between centers
    IS = [True, False, False, True, True, True]

    # centers throughputs
    lambdas = None
    # centers service rates
    mu = None
    # centers utilization factor
    rhos = None
    # centers average population
    nLocalAvgs = None
    nLocalAvgsOld = None
    # average permanence time for jobs to centers
    centersResponseTimeAvgs = None
    # system average response time for centers
    systemResponseTimeAvgs = None
    systemResponseTimeFromC1=None

    # global cycle time
    cycleTimeAvg = None
    # global throughput
    throughput = None

    probsFile = 'probs.pkl'

    # the threasholds probabilities array (to be red from file)
    thresholdProbabilities = None

    # the constructor
    def __init__(self,model, M,N,s1=31,s2=50):
        # loading the pribabilities file
        self.loadProbabilitiesFile()


        self.N =  N
        self.M = M
        self.s1 = s1
        self.s2 = s2

        # initializing threasholds probabilities
        self.ps1 = self.thresholdProbabilities[N-1][s1]
        self.ps2 = self.thresholdProbabilities[N-1][s2]

        self.vs = []
        self.lambdas = []
        self.mu = []
        self.rhos = []
        self.nLocalAvgs = []
        self.nLocalAvgsOld = []
        self.centersResponseTimeAvgs = []
        self.responseTimeAvg=0

        # compute the relative troughputs for each center
        # and store them into self.ys[][] matrix
        # if model is 1, then it's the original one, otherwise we consider the support model SM
```

```
        if model==1:
            # initializing the average service times for centers
            self.eTs=[7.0, 0.3, 0.08, 7.0, 0.05, 0.05]
            self.ysCompute1()
        else:
            # initializing the average service times for centers
            self.eTs=[7.0, 0.3, 0.08, 7.0, 7.05, 0.05]
            self.ysCompute2()

        # initializing global arrays
        for i in range (self.M):
            self.centersResponseTimeAvgs.append(self.eTs[i])
            self.nLocalAvgs.append(0)
            self.nLocalAvgsOld.append(0)
            self.vs.append([])
            self.lambdas.append(0)
            self.mu.append(1.0/self.eTs[i])
            for j in range (self.M):
                self.vs[i].append(float(self.ys[i])/self.ys[j])

    # load the threasholds probabilities file
    def loadProbabilitiesFile(self):
        self.thresholdProbabilities = pickle.load( open(self.probsFile,'rb') )

    # the mva recurring function for throughput
    def lambdaMVA(self,j,index):
        return (float(index+1) / self.lambdaSum(j))

    # computing the denominator for the lambda recurring function
    def lambdaSum(self,j):
        retval =0
        for i in range (self.M):
            retval += self.vs[i][j]* self.centersResponseTimeAvgs[i]
        return retval

    # the mva recurring function for the average population of center j
    def nAvg(self,j,lambdaJ):
        return lambdaJ*self.centersResponseTimeAvgs[j] #vj == one

    # the mva recurring function for the average response time of center j
    def tAvg (self,j):
        if self.IS[j]:
            return self.eTs[j]
        #if N is 0, then the nLocalAvgsOld is 0, so the tAvg is eTs
        return self.eTs[j]*(1.0+self.nLocalAvgsOld[j])

    # the mva recurring procedure
    def solve(self):
        for n in range (self.N):
            #Compute the response time for the center j
            for j in range (self.M):
                self.centersResponseTimeAvgs[j]= self.tAvg(j)
            #Compute the throughput of center j (lambda j)
            for j in range (self.M):
                self.lambdas[j]= self.lambdaMVA(j,n)
            #Compute the average of the number of job in the center j
            for j in range (self.M):
                self.nLocalAvgsOld[j]=self.nLocalAvgs[j]
                self.nLocalAvgs[j]= self.nAvg(j, self.lambdas[j])

        self.responseTimeAvgCompute()
```

```
    #----------------- local indexes --------------------
    #       (computing only the remaining ones)

    # compute the utilization factors for all centers
    def rhosCompute(self):
        for i in range (self.M):
            if self.IS[i]:
                self.rhos.append(0.0)
            else:
                self.rhos.append( float(self.lambdas[i] )* self.eTs[i])



    # ---------------- global indexes----------------------

    # compute the system response time for Client 1
    def responseTimeAvgGlobalCompute(self):
        self.systemResponseTimeClient1()

    # compute the system cycle time for Client 1
    def cycleTimeAvgGlobalCompute(self):
        self.cycleTimeAvg = self.responseTimeAvg + self.eTs[0]

    # compute the system throughput for Client 1
    def throughputCompute(self):
        self.throughput = float(self.N)/ self.cycleTimeAvg

    # compute the local indexes
    def localIndexesCompute(self):
        self.rhosCompute()


    #Compute the ys for the specified threasholds s1,s2
    def ysCompute1(self):

        self.ys = []
        # y1 == -6*(ps2 - 1)/((125*ps1 - 137)*ps2 - 363*ps1 + 375)
        self.ys.append( -6.0*(self.ps2 - 1.0)/((125.0*self.ps1 - 137.0)*self.ps2 - 363.0*self.ps1 + 375.0) )
        # y2 == 125*((ps1 - 1)*ps2 - ps1 + 1)/((125*ps1 - 137)*ps2 - 363*ps1 + 375)
        self.ys.append( 125.0*((self.ps1 - 1.0)*self.ps2 - self.ps1 + 1.0)/((125.0*self.ps1 - 137.0)*self.ps2 -
363.0*self.ps1 + 375.0) )
        # y3 == 125*((ps1 - 1)*ps2 - ps1 + 1)/((125*ps1 - 137)*ps2 - 363*ps1 + 375)
        self.ys.append( 125.0*((self.ps1 - 1.0)*self.ps2 - self.ps1 + 1.0)/((125.0*self.ps1 - 137.0)*self.ps2 -
363.0*self.ps1 + 375.0) )
        # y4 == -119*(ps1 - 1)/((125*ps1 - 137)*ps2 - 363*ps1 + 375)
        self.ys.append( -119.0*(self.ps1 - 1.0)/((125.0*self.ps1 - 137.0)*self.ps2 - 363.0*self.ps1 + 375.0) )
        # y5 == -6*(ps1*ps2 - ps1)/((125*ps1 - 137)*ps2 - 363*ps1 + 375)
        self.ys.append( -6.0*(self.ps1*self.ps2 - self.ps1)/((125.0*self.ps1 - 137.0)*self.ps2 - 363.0*self.ps1 + 375.0) )
        # y6 == -119*(ps1 - 1)*ps2/((125*ps1 - 137)*ps2 - 363*ps1 + 375)
        self.ys.append( -119.0*(self.ps1 - 1.0)*self.ps2/((125.0*self.ps1 - 137.0)*self.ps2 - 363.0*self.ps1 + 375.0) )


    #Compute the ys for the specified threasholds s1,s2
    def ysCompute2(self):

        self.ys = []
        # y1 == 6*((ps1 - 1)*ps2 - ps1 + 1)/((131*ps1 - 137)*ps2 - 369*ps1 + 375)
        self.ys.append( 6.0*((self.ps1 - 1.0)*self.ps2 - self.ps1 + 1.0)/((131.0*self.ps1 - 137.0)*self.ps2 - 369.0*self.ps1
```

```
+ 375.0) )
        # y2 == 125*((ps1 - 1)*ps2 - ps1 + 1)/((131*ps1 - 137)*ps2 - 369*ps1 + 375)
        self.ys.append( 125.0*((self.ps1 - 1.0)*self.ps2 - self.ps1 + 1.0)/((131.0*self.ps1 - 137.0)*self.ps2 -
369.0*self.ps1 + 375.0) )
        # y3 == 125*((ps1 - 1)*ps2 - ps1 + 1)/((131*ps1 - 137)*ps2 - 369*ps1 + 375)
        self.ys.append( 125.0*((self.ps1 - 1.0)*self.ps2 - self.ps1 + 1.0)/((131.0*self.ps1 - 137.0)*self.ps2 -
369.0*self.ps1 + 375.0) )
        # y4 == -119*(ps1 - 1)/((131*ps1 - 137)*ps2 - 369*ps1 + 375)
        self.ys.append( -119.0*(self.ps1 - 1.0)/((131.0*self.ps1 - 137.0)*self.ps2 - 369.0*self.ps1 + 375.0) )
        # y5 == -6*(ps1*ps2 - ps1)/((131*ps1 - 137)*ps2 - 369*ps1 + 375)
        self.ys.append( -6.0*(self.ps1*self.ps2 - self.ps1)/((131.0*self.ps1 - 137.0)*self.ps2 - 369.0*self.ps1 + 375.0) )
        # y6 == -119*(ps1 - 1)*ps2/((131*ps1 - 137)*ps2 - 369*ps1 + 375)
        self.ys.append( -119.0*(self.ps1 - 1.0)*self.ps2/((131.0*self.ps1 - 137.0)*self.ps2 - 369.0*self.ps1 + 375.0) )




    def globalIndexesCompute(self):
        self.responseTimeAvgGlobalCompute()
        self.cycleTimeAvgGlobalCompute()
        self.throughputCompute()

    def getExecutionTimeAvg(self,index):
        if index >= self.M:
            print 'error, index out of range'
            return None
        return self.centersResponseTimeAvgs[index]

    def getResponseTimeAvg(self,index):
        if index >= self.M:
            print 'error, index out of range'
            return None
        return self.systemResponseTimeAvgs[index]

    # aggiunto nuovo tempo di think presso c2 per i job rigettati da c2 (non necessario, opzione da scartare)
    def systemResponseTimeClient1(self):
        if self.systemResponseTimeFromC1:
            return self.systemResponseTimeFromC1
        self.systemResponseTimeFromC1 = 0.0
        for i in range (1,self.M):
            self.systemResponseTimeFromC1+= float(self.vs[i][0])* float(self.centersResponseTimeAvgs[i])
        return self.systemResponseTimeFromC1


    def responseTimeAvgCompute(self):
        self.systemResponseTimeAvgs = []
        for j in range (self.M):
            self.systemResponseTimeAvgs.append(0)
            for i in range (self.M):
                if i == j:
                    continue
                self.systemResponseTimeAvgs[j]+= self.vs[i][j]*self.centersResponseTimeAvgs[i]

    def getS1(self):
        return self.s1

    def getS2(self):
        return self.s2


    def printIndexes(self):
```

```
        print '=============================================='
        print '=\t\tLOCAL INDEXES\t\t='
        print '=============================================='
        for i in range (self.M):
            print '\ncenter %d:\n\tmu:\t\t\t%0.20f\n\tlambda:\t\t\t%0.20f\n\tutilizationFactor:\t%0.20f\n\tEn:\t\
            \t%0.20f\n\tEtr:\t\t\t%0.20f\n'%(i+1,self.mu[i],self.lambdas[i],self.rhos[i],\
                                            self.nLocalAvgs[i],self.systemResponseTimeAvgs[i])
        print '=============================================='
        print '=\t\tGLOBAL INDEXES\t\t='
        print '=============================================='
        print 'System Response Time:%f\nSystem Cycle Time:%f\nSystem Troughput:%f'%\
        (self.responseTimeAvg,self.cycleTimeAvg,self.throughput)
        print '\n\ndebug indexes:'
        print 's1:%d'%(self.s1)
        print 's2:%d'%(self.s2)
        print 'ps1:%f'%(self.ps1)
        print 'ps2:%f'%(self.ps2)
        print 'y:'
        for i in range (self.M):
            print '\ty%d:%f'%(i,self.ys[i])
        print 'thr size:%d'%(len(self.thresholdProbabilities[self.N-1]))


    def printFirstIndexes(self):

        equivLambda = -self.lambdas[4]
        rho = self.lambdas[0]/self.mu[0]
        equivN = self.nLocalAvgs[0]+self.nLocalAvgs[4]
        print '\ncenter client 1:\n\tmu:\t\t\t%f\n\tlambda:\t\t\t%f\n\tutilizationFactor:\t%f\n\tEn:\t\
        \t%f\n\tEtr:\t\t\t%f\n'%(self.mu[0],equivLambda,rho, equivN,self.systemResponseTimeClient1())
        print 's1:%d'%(self.s1)
        print 's2:%d'%(self.s2)
        print 'ps1:%f'%(self.ps1)
        print 'ps2:%f'%(self.ps2)
        print 'y:'
        for i in range (self.M):
            print '\ty%d:%f'%(i,self.ys[i])


    def printVs(self):
        print 'v:'
        for i in range(self.M):
            for j in range(self.M):
                print '\tv[%d][%d]:%f'%(i,j,self.vs[i][j])

    def printLambdas(self):
        print 'lambdas:%s'%(str(self.lambdas))

    def makeCsv(self, filename):
        path='%s.csv'%(filename)
        f=open(path,"w")
        f.write('customer number, threshold S1, threshold S2,\n%d,%d,%d,\n\n,'%(self.N,self.s1,self.s2))
        for i in range(self.M):
            f.write('center %d,'%(i+1))
        f.write('center client1 (with reject),')
        f.write('\nthroughputs,')
        for i in range(self.M):
            f.write('%f,'%(self.lambdas[i]))
        f.write('%f,'%(float(self.lambdas[0])-self.lambdas[4]))

        f.write('\nresponse times,')
```

```
        for i in range(self.M):
            f.write('%f,'%(self.centersResponseTimeAvgs[i]))
        # the infinite server response time is the service time
        f.write('%f,'%(self.centersResponseTimeAvgs[0]+self.centersResponseTimeAvgs[4]*self.ps1))

        f.write('\nsystem response times,')
        for i in range(self.M):
            f.write('%f,'%(self.systemResponseTimeAvgs[i]))
        f.write('%f,'%(self.systemResponseTimeClient1_2()))

        f.write('\ncycle times,')
        for i in range(self.M):
            f.write('%f,'%(self.systemResponseTimeAvgs[i]+self.centersResponseTimeAvgs[i]))
        f.write('%f,'%(self.systemResponseTimeClient1_2()+
self.centersResponseTimeAvgs[0]+self.centersResponseTimeAvgs[4]*self.ps1))


        f.close()

    def responseTimesToString(self):
        retval = ''
        for i in range(self.M):
            retval+='%f,'%(self.centersResponseTimeAvgs[i])
        return retval

    def populationAvgsToString(self):
        retval = ''
        for i in range(self.M):
            retval+='%f,'%(self.nLocalAvgs[i])
        return retval

    def throughputAvgsToString(self):
        retval = ''
        for i in range(self.M):
            retval+='%f,'%(self.lambdas[i])
        return retval

    def utilizationAvgsToString(self):
        retval = ''
        for i in range(self.M):
            retval+='%f,'%(self.nLocalAvgs[i]*self.centersResponseTimeAvgs[i])
        return retval

    def responseTimesToFile(self,dstFile,toWrite):
        dstFile.write('%s'%toWrite)
        for i in range(self.M):
            dstFile.write('%f,'%(self.centersResponseTimeAvgs[i]))
        # the infinite server response time is the service time
        dstFile.write('%f,\n'%(self.centersResponseTimeAvgs[0]+self.centersResponseTimeAvgs[4]*self.ps1))


    def systemResponseTimesToFile(self, dstFile,toWrite):
        dstFile.write('%s'%toWrite)
        for i in range(self.M):
            dstFile.write('%f,'%(self.systemResponseTimeAvgs[i]))
        dstFile.write('%f,\n'%(self.systemResponseTimeClient1_2()))

    def shortSystemResponseTimesToFile(self, dstFile,toWrite):
        dstFile.write('%s'%toWrite)
        dstFile.write('%f,\n'%(self.systemResponseTimeClient1_2()))
```

```
    def sageWrite(self,dstFile,toWrite):
        dstFile.write('%s'%toWrite)
        dstFile.write('%f,\n'%(self.systemResponseTimeClient1_2()))


# the main function for mva
if __name__=='__main__':
    m = 6
    cond = False

    while True:
        print '-------------------------------------------------'
        times = []
        s = raw_input('inserire n:')
        n = int(s)
        s = raw_input('inserire s1:')
        s1 = int(s)
        s = raw_input('inserire s2:')
        s2 = int(s)

        times.append(datetime.datetime.now())

        mmva = MVA(m,n,s1,s2)

        mmva.solve()
        mmva.localIndexesCompute()
        mmva.globalIndexesCompute()
        times.append(datetime.datetime.now())

        mmva.printIndexes()
        mmva.printFirstIndexes()
        mmva.makeCsv('testN-%s_S1-%d_S2-%d'%(n,s1,s2))
        mmva.debug2()
```

BIBLIOGRAFIA

[1] “An analytical model for multi-tier internet services and its applications” B. Urgaonkar , G. Pacifici , P. Shenoy , M. Spreitzer , A. Tantawi
[2] Giuseppe Iazeolla- Impianti Reti Sistemi Informatici-*Università degli Studi di Roma Tor Vergata*
[3] Dispense corso di Modelli di Prestazioni di Sistemi e Reti- Prof.ssa Vittoria De Nitto Personé