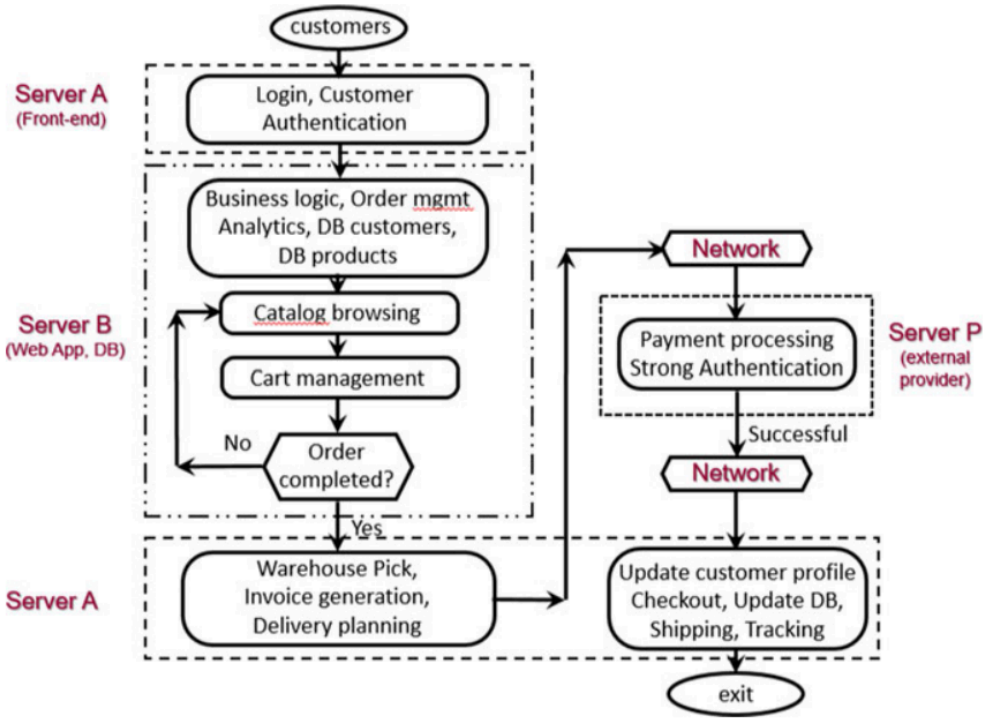
1. Introduzione

Si analizza il workflow di una web app di un e-commerce. Il sistema è composto da tre server:

- server A, il quale si occupa dell'autenticazione dei customers, interazione con il processo di pagamento, generazione fattura elettronica, operazioni di checkout sul DB, invoice generation, servizi di tracking e shipping, aggiornamento dei dati degli utenti;
- server B, che si occupa del core business, mantiene un database con dati sui prodotti e clienti, permette di effettuare browsing del catalogo e processamento dello shopping cart;
- server P, che si occupa del pagamento. Questo server appartiene ad un provider esterno.
  L'interazione con la web app viene modellata come una sequenza di azioni sui server del sistema, in particolare ogni richiesta prevede due visite al server A, una visita al server B e una visita al server P seguendo la seguente sequenza: $A \rightarrow B \rightarrow A \rightarrow P \rightarrow A$.
  Al server B viene poi data la possibilità di replicarsi in caso di alto carico.



1.2 Obiettivi del caso di studio

Il presente caso di studio si basa sul caso di studio "Simulation of the Workflow of a Web App" del libro di testo "Performance Engineering".
Come primo step si modella il sistema considerando il tasso di arrivi pari a 1800 req/h (0.5 req/s) con i service demands presenti nella parte sinistra della tabella 1.
Anche gli obiettivi seguono quelli presenti sul libro di testo con aggiunta dell'obiettivo 4. Gli obiettivi dunque sono i seguenti:

- obiettivo 1, analisi delle prestazioni con tasso di arrivi pari a 4320 req/h (1.2 req/s);
- obiettivo 2 (F2A), analisi dell'impatto del sistema di autenticazione a due fattori (service demand della parte destra tabella 1);
- obiettivo 3 (SCALING), analisi delle prestazioni introducendo lo scaling orizzontale per il server B, con tasso di arrivi pari a 4320 req/h.

2. Conceptual Model

Per distinguere le fasi dei job all'interno del sistema, sono state definite tre classi:

- classe 1, job che arrivano dall'esterno ($\rightarrow A$) e job che passano dal server A ad una copia del server B ($A \rightarrow B$);
- classe 2, job che arrivano da una copia del server B al server A ($B \rightarrow A$) e dal server A al server P ($A \rightarrow P$);
- classe 3, job che arrivano dal server P verso il server A ($P \rightarrow A$).

Quindi j-esimo job appartiene ad una classe indicata con la variabile $C_j(t) \in \{1, 2, 3\}$ dove t indica l'istante di tempo attuale.

Per i server sono definite le seguenti variabili di stato:

- $N_A^c(t)$ con $c \in \{1, 2, 3\}$, numero di job di classe c nel nodo A all'istante t;
- $N_B(t)$, numero di job nel nodo B (nel nodo B ci possono essere solo job della classe 1);
- $B(t)$, numero di copie del server B attive all'istante t (valore minimo pari ad 1);
- $N_P(t)$, numero di job nel nodo P (nel nodo P ci possono essere solo job della classe 2);
- $N(t)$, numero di job nel sistema all'istante t.

Nel caso di studi non sono considerati ritardi dovuti alla rete che collega i server e non è considerato il think time degli utenti dell'e-commerce; dunque, non sono presenti variabili che modellano questi aspetti.

3 Specification Model

Per il caso di studio non sono disponibili dati da analizzare per ottenere un input model; quindi, tutti i dati e valori che seguono sono presi dal libro di testo.
Per il contenuto di questo paragrafo sono stati presi in considerazioni i tempi medi di servizio riportati nel caso di studi del libro.

Stations	Classes		
	1	2	3
Server A (Login, Front end, ...)	0.2	0.4	0.1
Server B (Web App Serv., DBs, ...)	0.8	0	0
Server P (Payment Provider)	0	0.4	0

Stations	Classes		
	1	2	3
Server A (Login, Front end, ...)	0.2	0.4	0.15
Server B (Web App Serv., DBs, ...)	0.8	0	0
Server P (Payment Provider)	0	0.7	0

Tabella 1.

3.1.1 Obiettivo 1

Come distribuzione di probabilità per gli arrivi esterni è stata scelta una esponenziale $Exp(\lambda)$.
Tasso arrivi $\lambda = 1.2\ req/s\ (4320\ req/h)$.
Come politica di scheduling per tutti gli obiettivi è stato scelta Processor Sharing.

Per le distribuzioni di probabilità dei tempi di servizio sono state scelte delle esponenziali $Exp(\frac{1}{E[s]})$.
Server A:

- tempo medio servizio classe 1, $0.2s$;
- tempo medio servizio classe 2, $0.4s$;
- tempo medio servizio classe 3, $0.1s$.
  Server B: tempo medio servizio (classe 1) $0.8s$.
  Server P: tempo medio servizio (classe 2) $0.1s$.

3.1.2 Obiettivo 2 (F2A INTRODUCTION)

Come distribuzione di probabilità per gli arrivi esterni è stata scelta una esponenziale $Exp(\lambda)$.
Tasso arrivi $\lambda = 1.2\ req/s\ (4320\ req/h)$.

Per le distribuzioni di probabilità dei tempi di servizio sono state scelte delle esponenziali $Exp(\frac{1}{E[s]})$.
Server A:

- tempo medio servizio classe 1, $0.2s$;
- tempo medio servizio classe 2, $0.4s$;
- tempo medio servizio classe 3, $0.15s$.
  Server B: tempo medio servizio (classe 1) $0.8s$.
  Server P: tempo medio servizio (classe 2) $0.7s$.

3.1.3 Obiettivo 4 (SCALING)

Come distribuzione di probabilità per gli arrivi esterni è stata scelta una esponenziale $Exp(\lambda)$.
Tasso arrivi $\lambda = 1.2\ req/s\ (4320\ req/h)$.

Per le distribuzioni di probabilità dei tempi di servizio sono state scelte delle esponenziali $Exp(\frac{1}{E[s]})$.
Server A:

- tempo medio servizio classe 1, $0.2s$;
- tempo medio servizio classe 2, $0.4s$;
- tempo medio servizio classe 3, $0.15s$.
  Server B: tempo medio servizio (classe 1) $0.8s$.
  Server P: tempo medio servizio (classe 2) $0.7s$.

4. Computational Model

Per la realizzazione del codice è stato scelto il linguaggio Java per il modello computazionale. Per la generazione dei grafici e per implementare altre funzioni di ausilio è stato scelto Python.
Tutto il codice è salvato nella repository Github: https://github.com/jack-mack15/ProgettoPMCSN.

4.1 Next Event Scheduler

La classe NextEventScheduler implementa tutta la logica di registrazione, smistamento e salvataggio degli eventi.
Possiede una lista di eventi implementata con una PriorityQueue, struttura dati di Java che ordina gli elementi in base un campo. In questo modo le operazioni sulla lista automaticamente riordinano la lista stessa in base all'istante di arrivo degli eventi. Per la corretta gestione di eventi che si verificano nello stesso istante, si da priorità agli eventi DEPARTURE, CREATE e DESTROY.
Questa classe espone un metodo per aggiungere eventi alla lista, in questo modo si può emulare l'invio di un job ad un altro server tramite inserimento di evento ARRIVAL nella lista di eventi.
Altro metodo importante è il metodo che permette di eliminare un evento dalla lista di eventi. Questo verrà analizzato in Server.
La logica principale è implementata nel metodo run() che si occupa di ciclare fino a che non si raggiunge il limite di fine simulazione. Ad ogni ciclo estrae il primo elemento per istante di arrivo dalla lista, verifica il tipo dell'evento estratto e invoca il corretto gestore dell'evento.
Tutti gli eventi tranne SAMPLING vengono inviati alla classe System, la quale indirizza al corretto server l'evento attuale. Invece l'evento SAMPLING viene inviato alla classe Statistic, la quale si occupa di effettuare i campionamenti.
Altro importante compito di questa classe è la gestione del clock di simulazione. Per il clock, sono esposti solo metodi di lettura. La modifica del clock è riservata esclusivamente alla

classe NextEventScheduler. Tali modifiche avvengono quando si estrae un nuovo evento dalla lista e al clock viene assegnato l'istante di arrivo dell'evento appena estratto.

4.2 Eventi

La simulazione del caso di studio è di tipo next event basata su due quattro tipi di eventi:

- ARRIVAL, arrivo di un job in un nodo. Si incrementa il numero di job del server;
- DEPARTURE, partenza di un job da un nodo. Si decrementa il numero di job del server e viene instradato il job verso il successivo server;
- CREATE, creazione copia server B e si incrementa il numero di copie di B;
- DESTROY, rimozione copia server B e si decrementa il numero di copie di B.
  Oltre a questi eventi è stato definito un evento artificiale SAMPLING necessario per ottenere dati sui server e sul sistema.
  Per il server A la gestione dei seguenti eventi differisce dagli altri server:
- ARRIVAL, viene incrementato il numero di job totali nel nodo A e il numero di job nel nodo A della stessa classe del nuovo job. Se l'arrivo è dall'esterno si assegna classe 1 al job, si procede con il generare il nuovo arrivo da esterno e si incrementa il numero di job nel sistema;
- DEPARTURE, viene decrementato il numero di job nel nodo A e il numero di job nel nodo A della stessa classe del job corrente. A seconda della classe del job appena processato, si cambia la classe. Se il job associato all'evento è di classe 2, allora il job esce dal sistema e viene decrementato il numero di job nel sistema.

Per gestire correttamente gli eventi, la classe Event possiede gli attributi: time, istante di arrivo dell'evento; type, uno dei 5 tipi di eventi descritti sopra; node, stringa che riporta il destinatario dell'evento; classId, identificatore della classe (1,2 o 3); idRequest, identificatore univoco dell'evento.

4.3 Job

La classe job contiene tutte le informazioni per la sua corretta gestione e tutte le metriche interessanti al caso di studi. Alcuni attributi come id, jobClass e node sono simili alla classe Event. Gli altri attributi sono: arrivalTime, istante di arrivo, serviceTime, tempo di servizio richiesto; remainServiceTime, tempo di servizio rimanente; completeTime, tempo di completamento.
Gli identificatori di un job corrispondono agli identificatori degli eventi associati. Quindi tutte le istanze di Job e istanze di Event dello stesso job logico che vengono processati dai server, posseggono stesso identificatore.
Inoltre, la classe Job espone anche i metodi necessari per gestire il tempo di servizio richiesto e rimanente dei job. In particolare, la classe Job riceve lo share di tempo calcolato dai server e aggiorna il remainServiceTime dell'istanza corrente.

4.4 System, Server e Load Balancer

4.4.1 System

La classe System è l'intermediario tra la classe NextEventScheduler e le classi dei Server. Mantiene traccia di quali sono i server e smista ai corretti server gli eventi che riceve dallo scheduler.

4.4.2 LoadBalancer

Questa è la classe che si occupa di gestire le copie del server B in caso di modello SCALING e la gestione degli eventi DESTROY e CREATE. L'evento DESTROY viene generato dal server B stesso, ma la gestione è eseguita dal LoadBalancer.
Dunque, questa classe ha una lista di copie di server B. Quando è necessaria una nuova copia, la crea assegnando anche un identificatore alla copia. La rimozione sfrutta proprio questo identificatore.
Il LoadBalancer si occupa anche di scegliere quale copia di B deve ricevere e gestire l'evento ARRIVAL o DEPARTURE. La selezione in caso di arrivo scorre la lista di copie e verifica quanti job sono presenti nella copia, quindi la ricerca è First Fit. Se non trova nessuna copia libera, genera l'evento CREATE, il quale arriva allo scheduler e poi torna al LoadBalancer stesso.
Ultima importante funzionalità di questa classe è il conteggio dei job scartati (per la fase di verifica modello SCALING). Nella selezione della copia che deve gestire un arrivo, se non trova una copia libera e il numero di copie è già massimo, il job viene scartato e vengono incrementati i contatori associati.

4.4.3 Server

I server sono la componente più complessa del modello poichè gestiscono gli eventi ARRIVAL e DEPARTURE, e gestiscono anche il tempo di servizio dei job. Sono implementati con le seguenti classi: NodeA, NodeB e NodeP. Tutte queste classi implementano la corretta gestione degli eventi, con piccole modifiche a seconda del server.

Quando arriva un evento ARRIVAL, viene eseguito il metodo onArrival(). Questo metodo si occupa di creare una istanza di Job assegnando come campi il tempo di arrivo, id e classe dell'evento. Il tempo di servizio del job viene generato con la classe Exponential (ogni server ha il proprio stream). Se l'arrivo è sul server A, la gestione è leggermente più complessa, poiché occorre capire a quale classe appartiene il job e di conseguenza agire in modo leggermente differente. Una volta che l'istanza di Job è stata creata, si aggiorna la popolazione del server, si aggiorna il tempo di servizio rimanente degli altri job già presenti, si aggiunge il nuovo job alla lista di job e si genera il prossimo evento DEPARTURE per il server stesso.

In caso di evento DEPARTURE, viene eseguito il metodo onDeparture(). Il primo step è quello di aggiornare il tempo di servizio rimasto dei job, verificare se ci sono job con tempo di servizio rimanente esaurito.
In caso ci siano job completati, questi devono essere inviati al successivo server. la logica di questa azione è implementata dal metodo sendJobToServer() che a seconda del server in

cui mi trovo, genero un evento DEPARTURE con le informazioni corrette.
Infine, il metodo onDeparture() si occupa di generare l'evento DEPARTURE successivo per sé stesso. Per fare questo si prende il job con tempo di servizio rimanente minore, e si divide tale tempo per il numero di job presenti sul server (share di tempo). Quindi il prossimo evento di partenza avrà istante di arrivo il tempo corrente più lo share calcolato.
Attenzione: per ogni server deve esistere al più un evento DEPARTURE per garantire la corretta gestione del tempo di servizio. Dunque, ogni server, all'arrivo di evento di partenza o di arrivo, chiede allo scheduler di rimuovere eventuali (situazione che non dovrebbe verificarsi) eventi DEPARTURE per sé stesso dalla lista di eventi.

Per gestire correttamente il tempo di servizio, è stato implementato il metodo updateRemainingServiceTime(). Questo metodo viene invocato quando arriva un evento di partenza o di arrivo. Si calcola lo share di tempo (per implementare logica processor scheduling) con intervallo di tempo dall'ultimo aggiornamento ad istante corrente e numero di job presenti.

4.5 Simulation Controller

SimulationController è la classe che gestisce le simulazioni e corrisponde al main() del codice. Da questa classe è possibile configurare ogni parametro del sistema e degli esperimenti.

4.6 Pseudo random number generator

Come PRNG è stato utilizzato un Lehmer generator a 32 bit seguendo le indicazioni viste nel corso. I parametri utilizzati sono i seguenti:

- modulo $m = 2147483647$;
- moltiplicatore $a = 48271$.
  Sono stati generati 256 streams tramite il jump multiplier $j = 22925$. In questo modo per ogni stream si ha a disposizione 8367782 numeri da generare. Gli stream sono organizzati in un array di 256 elementi, che inizialmente corrispondono ai seed degli stream. Generare un valore comporta anche modificare il valore salvato nell'array (ad indice corrente).
  Il numero di streams permette di avere uno stream differente per componente del sistema e streams differenti tra repliche differenti. Una singola replica infatti richiede 6 streams differenti. Per la selezione dello stream corretto viene applicato un offset fisso ad un indice specifico della run. Questo indice della run viene incrementato di 6 ad ogni replica. Gli offset per la selezione degli stream sono:
- 0, stream per gli arrivi dall'esterno;
- 1, stream per i tempi di esecuzione dei job arrivati dall'esterno nel server A;
- 2, stream per i tempi di esecuzione dei job di classe 1 nel server A;
- 3, stream per i tempi di esecuzione dei job di classe 2 nel server A;
- 4, stream per i tempi di esecuzione dei job del server B;

- 5, stream per i tempi di esecuzione dei job nel server P.
  La classe $NextEventScheduler$, prima di avviare le simulazioni, si occupa della creazione di un array di 256 elementi che corrispondono ai seed iniziali degli streams a disposizione. Il seed di partenza da cui vengono poi generati tutti gli streams può essere scelto dall'utente altrimenti verrà utilizzato il valore default 123456789.

Il codice del generatore è quello fornito dal libro di testo, al quale sono state apportate delle leggere modifiche solo alla visibilità dell'array degli stream e accesso ad essi. In questo modo i seed vengono generati una sola volta per esecuzione di modello.

4.7 Arrival Controller

Questa è la classe che si occupa di gestire la generazione degli eventi ARRIVAL. Il metodo principale è generateExtArrival() che viene invocato o ad inizio simulazione oppure ad ogni gestione da parte dello scheduler di un evento ARRIVAL. Gli eventi generati hanno sempre come server obiettivo il server A.

4.8 Metriche del sistema

Le metriche scelte per il sistema sono:

- popolazione media dei nodi e del sistema;
- utilizzazione dei nodi;
- throughput dei nodi e del sistema;
- waiting time dei nodi;
- tempo di risposta dei nodi e del sistema.

La raccolta delle metriche è affidata alla classe Statistic. Questa gestisce l'evento SAMPLING. Il comportamento di questa classe cambia a seconda del tipo di esperimento; infatti, invoca i metodi di due classi: ClassicStatistic e BatchMeansEstimator.
La classe ClassicStatistic si occupa dell'aggiornamento e campionamento per generare i grafici temporali. Quindi quando si presenta un evento SAMPLING, vengono campionate tutte le metriche di interesse e vengono scritte nel file associato.
La classe BatchMeansEstimator si occupa di gestire l'aggiornamento delle metriche per implementare il metodo Batch Means. Quindi ad ogni completamento di job aggiorna le metriche del batch e quando il numero di completamente raggiunge la size del batch, viene scritta una riga nel file associato con il valore delle metriche e il batch (con tutte le metriche) viene resettato.
La classe ClassicStatistic gestisce le metriche di tutti i server e sistema complessivo contemporaneamente, mentre la classe BatchMeansEstimator lo fa in modo separato grazie ad istanze della classe BatchClass (4 istanze, una per ogni server e una per sistema). Dunque, l'aggiornamento delle metriche e scrittura su file per il caso del Batch Means viene fatto per componente attuale.

5. Verifica

Per la fase di verifica sono state confrontate le tre versioni del modello computazionale (base, F2A e modello con scaling orizzontale) con i corrispondenti modelli analitici.
Per ottenere il tempo medio di servizio e utilizzazione del server A si sono considerate le seguenti formule:

- $E(S_A) = \sum_{c=1}^{3} E(S_{A,c}) p_{A,c}$
- $\rho_A = \lambda_A E(S_A)$
  Dove $p_{A,c}$ rappresenta la probabilità di trovare job della classe c ($c \in \{1, 2, 3\}$) nel server A e $E(S_{A,c})$ rappresenta il tempo medio di servizio nel server A della classe c. Per λ_A vedere sotto.
  Nel modello analitico, per il tempo di risposta medio e popolazione media dei singoli server, sono state utilizzate le seguenti formule:
- $E[T]_s = \frac{E(S_s)}{1-\rho_s} = \frac{1}{\mu_s - \lambda_s}$
- $E[N]_s = E[T]_s \lambda_s$
  Invece per gli indici globali sono state utilizzate le seguenti formule:
- $v_s = \frac{\lambda_s}{\lambda}$
- $E[T] = \sum_{s \in \{A,B,P\}} E[T]_s v_s$
- $E[N] = \lambda E[T]$
  Per ottenere i valori di λ_i è stata utilizzata la formula:
- $\lambda_i = \gamma_i + \sum_{j=1}^{k} \lambda_j p_{ji}$
  Per ogni server si ottiene una espressione del proprio λ che porta ad un sistema di 3 equazioni con 3 incognite, da cui si sono ottenuti i seguenti valori:

  $$\begin{cases} \lambda_A = 3\lambda \\ \lambda_B = \lambda \\ \lambda_P = \lambda \end{cases}$$

  Nel caso della verifica del modello SCALING, tali valori cambieranno a causa della probabilità di perdita:

  $$\begin{cases} \lambda_A = \lambda + 2\lambda_{eff} \\ \lambda_B = \lambda \\ \lambda_P = \lambda_{eff} \end{cases}$$

  con $\lambda_{eff} = \lambda(1 - P_{loss})$.
  Per la popolazione media del server B nella verifica del modello SCALING è stata utilizzata la formula:
- $E[N] = \sum_{i,j=0}^{3} (i+j)\pi_{i,j}$

Per gli intervalli di confidenza sono state utilizzate 64 repliche indipendenti eseguite per una finestra temporale di 48 ore.

5.1 Modello Base

In questo modello i tempi medi di esecuzione dei server fanno riferimento ad obiettivo 1 (section 3.1.1).

Non sono previste copie del server B e il tasso di arrivi dall'esterno è pari a $1.2\ req/s$.
Dal modello analitico si ottengono gli indici globali:

- mean response time pari a 25.1381 secondi;
- mean population pari a 30.176 job.
  Dal modello computazionale si ottengono i seguenti indici globali:
- mean response time pari a 25.0766 ± 0.5365;
- mean population pari a 30.0947 ± 0.6547.

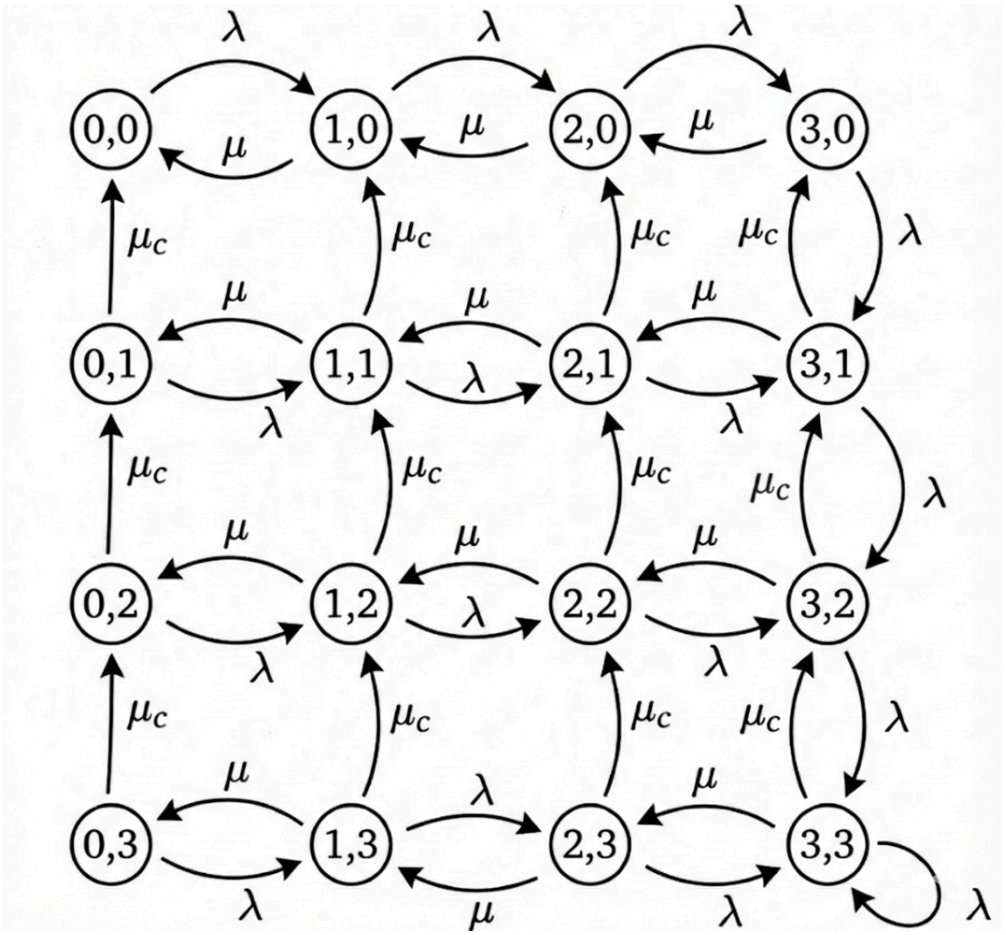
5.2 Modello F2A

In questo modello i tempi medi di esecuzione dei server fanno riferimento ad obiettivo 1F2A (section 3.1.2).
Non sono previste copie del server B e il tasso di arrivi dall'esterno è pari a $1.2\ req/s$.
Dal modello analitico si ottengono gli indici globali:

- mean response time pari a 37.5 secondi;
- mean population pari a 38.250 job.
  Dal modello computazionale si ottengono gli indici globali:
- mean response time pari a 31.7605 ± 0.5487;
- mean population pari a 38.1141 ± 0.6731.

Modello F2A con Scaling

In questo modello i tempi medi di esecuzione dei server fanno riferimento ad obiettivo SCALING (section 3.1.3).
Sono previste copie del server B che vengono create in modo dinamico e il tasso di arrivi dall'esterno è pari a $1.2\ req/s$.
Per la verifica di questo modello è stata identificata una Markov Chain dove con numero di copie massimo pari a 2 e numero di job massimi per copia pari a 3.
Dunque, uno stato (i,j) identifica il numero di job presenti in copia uno e copia due (con i,j <= 3). Il tasso di arrivi è identificato con λ, tasso di servizio della prima copia μ e tasso di servizio della seconda copia μ_c. I due tassi di servizio sono identici, ma sono stati differenziati per distinguere il passaggio da uno stato ad un altro in base a quale copia termina.
La catena di Markov è la seguente.



Dalla catena di Markov (con equazioni di bilancio flusso) sono state individuate le seguenti probabilità di stato:

- $\pi_0 = \frac{\mu}{\lambda}\pi_1 + \frac{\mu}{\lambda}\pi_{0,1}$
- $\pi_1 = \frac{\lambda}{\lambda+\mu}\pi_0 + \frac{\mu}{\lambda+\mu}(\pi_{1,1} + \pi_2)$
- $\pi_2 = \frac{\lambda}{\lambda+\mu}\pi_1 + \frac{\mu}{\lambda+\mu}(\pi_3 + \pi_{2,1})$
- $\pi_3 = \frac{\lambda}{\lambda+\mu}\pi_2 + \frac{\mu}{\lambda+\mu}\pi_{3,1}$
- $\pi_{3,1} = \frac{\lambda}{\lambda+2\mu}(\pi_{2,1} + \pi_3) + \frac{\mu}{\lambda+2\mu}\pi_{3,2}$
- $\pi_{3,2} = \frac{\lambda}{\lambda+2\mu}(\pi_{2,2} + \pi_{3,1}) + \frac{\mu}{\lambda+2\mu}\pi_{3,3}$
- $\pi_{3,3} = \frac{\lambda}{2\mu}(\pi_{2,3} + \pi_{3,2})$
- $\pi_{2,3} = \frac{\lambda}{\lambda+2\mu}\pi_{1,3} + \frac{\mu}{\lambda+2\mu}\pi_{3,3}$
- $\pi_{1,3} = \frac{\lambda}{\lambda+2\mu}\pi_{0,3} + \frac{\mu}{\lambda+2\mu}\pi_{2,3}$
- $\pi_{0,3} = \frac{\mu}{\lambda+\mu}\pi_{1,3}$
- $\pi_{0,2} = \frac{\mu}{\lambda+\mu}(\pi_{0,3} + \pi_{1,2})$
- $\pi_{0,1} = \frac{\mu}{\lambda+\mu}(\pi_{0,2} + \pi_{1,1})$
- $\pi_{1,1} = \frac{\lambda}{\lambda+2\mu}\pi_{0,1} + \frac{\mu}{\lambda+2\mu}(\pi_{2,1} + \pi_{1,2})$
- $\pi_{2,1} = \frac{\lambda}{\lambda+2\mu}\pi_{1,1} + \frac{\mu}{\lambda+2\mu}(\pi_{3,1} + \pi_{2,2})$
- $\pi_{2,2} = \frac{\lambda}{\lambda+2\mu}\pi_{1,2} + \frac{\mu}{\lambda+2\mu}(\pi_{2,3} + \pi_{3,2})$
- $\pi_{1,2} = \frac{\lambda}{\lambda+2\mu}\pi_{0,2} + \frac{\mu}{\lambda+2\mu}(\pi_{1,3} + \pi_{2,2})$

  Per la risoluzione di tali equazioni è stato implementato un codice in Python (eseguito su Google Colab) che utilizza la libreria Numpy. Le probabilità calcolate sono le seguenti:

Stato	Probabilità
0	0.24852
01	0.01087
02	0.00466
03	0.00147
1	0.22770
11	0.01665
12	0.00766
13	0.00287
2	0.19107
21	0.03119
22	0.01533
23	0.0071
3	0.12472
31	0.06101
32	0.03093
33	0.01825

L'ultima probabilità, stato 33, corrisponde proprio alla probabilità di perdita.
Ulteriori valori del server B calcolati con il modello analitico sono:

- throughput effettivo pari a 1.17810 req/s;
- numero utenti medio pari a 1.77494;
- tempo di risposta medio pari a 1.50661 s.
  Dal modello computazionale si ottengono:
- tempo di risposta medio 1.5029 ± 0.0045;
- numero medio di utenti 1.7671 ± 0.0061;
- throughput effettivo 1.758 ± 0.0021;
- probabilità di loss 0.01976 ± 0.00048.

6. Validazione

Non avendo dati di una reale implementazione del caso di studio non è stato possibile effettuare la validazione del modello computazionale.

7. Transitorio

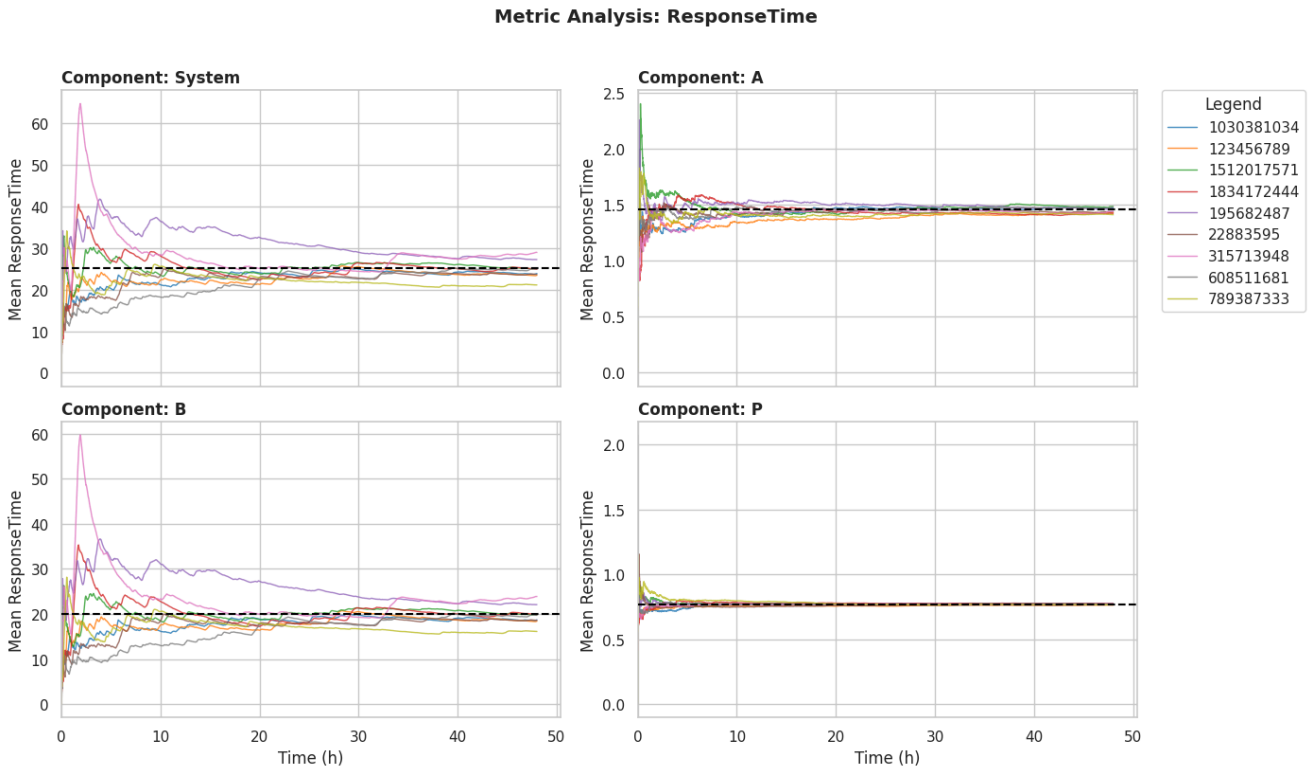
In questa sezione si vuole verificare se il sistema nelle varie configurazioni previste converge. Per ogni configurazione si eseguono 9 run (ciascuna con seed iniziale differente)

per una finestra temporale di 48 ore. Dalle quattro successive sezioni, in cui si mostrano i risultati per le quattro configurazioni del modello computazionale che solo il caso di OVERLOAD con tasso arrivi 1.4 req/s non converge. In tal caso il Server B non riesce a gestire il traffico; dunque, popolazione media e tempo di risposta medio aumentano linearmente. L'utilizzazione del Server B raggiunge immediatamente utilizzazione pari ad 1. A causo di questi incrementi lineari, anche le statistiche globali riportano gli stessi incrementi lineari.

7.1 Transitorio modello base

La prima configurazione prevede il sistema senza F2A e tasso di arrivi pari a 1.2 req/s. Per dimostrarla convergenza del sistema e dei server, nei grafici riportati è presente una retta tratteggiata nera che mostra il valore restituito dal modello equivalente. In particolare, le rette hanno equazioni:

- y = 1.4592 per A;
- y = 20 per B;
- y = 0.7692 per P;
- y = 25.1381 per System.
  Il tempo di risposta medio di tutte le componenti converge. I server A e P sono più rapidi poiché hanno utilizzazione più bassa come si può vedere dal grafico dell'utilizzazione, mentre il server B ha un primo periodo molto instabile. Il comportamento del sistema, globalmente, è influenzato dal server B, dunque gli andamenti sono molto simili.

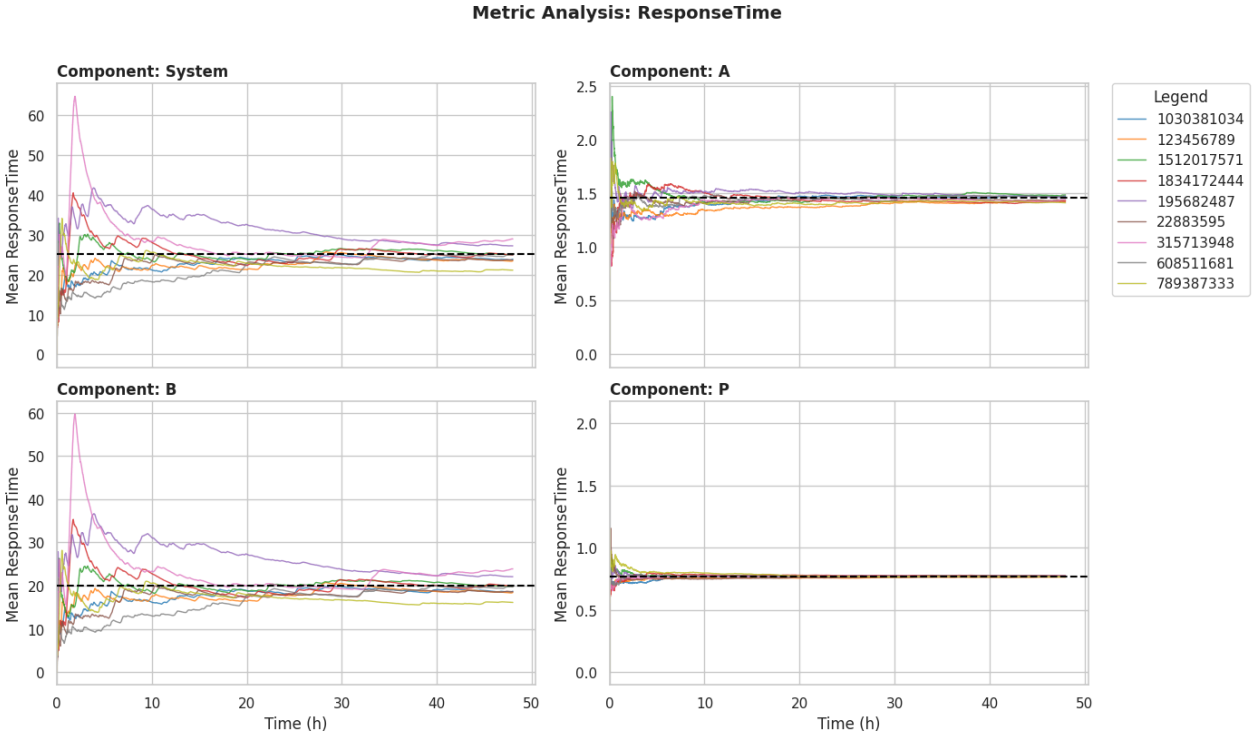


7.2 Transitorio modello con F2A

In questa sezione il modello prevede l'autenticazione a due fattori e dunque sono incrementati i tempi medi di servizi dei server A e P. Il tasso di arrivi rimane 1.2 req/s e

l'output prodotto è simile al caso precedente, dove le rette hanno valori differenti. In particolare:

- y = 2.5 per A;
- y = 20 per B;
- y = 4.375 per P;
- y = 37.5 per System.
  Il tempo di risposta medio per i server A e P aumenta rispetto al modello base ma anche qui per tutte le componenti converge.

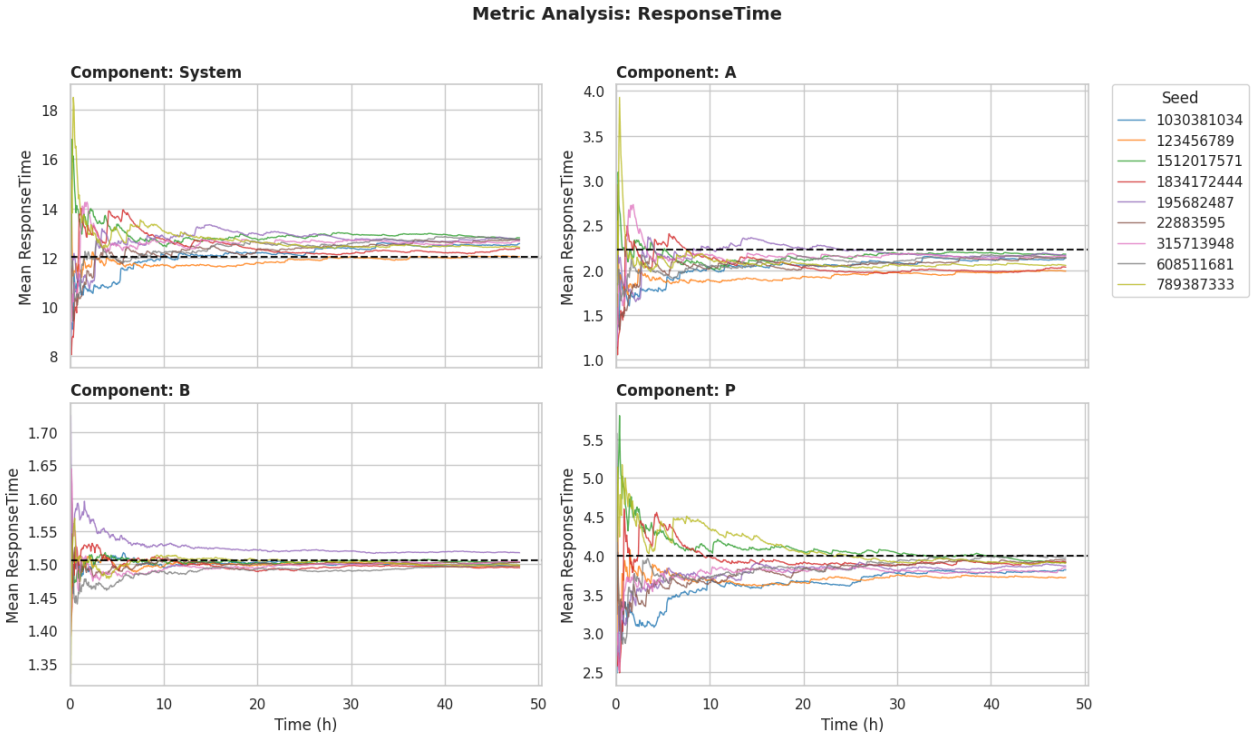


7.3 Transitorio per SCALING

Adesso il modello prevede scaling orizzontale, quindi i parametri di tempi medi di servizio e tasso di arrivi corrispondono al caso F2A ma Server B ha la possibilità di creare copie di sé stesso.
La configurazione dello scaling è la stessa della fase di verifica. Questo comporta che Server B rifiuterà job in caso entrambe le copie del server siano piene, impattando non solo le prestazioni del server B ma anche il throughput verso i server A e P. Dunque, le rette di riferimento in questo caso sono:

- y = 2.228477 per A;
- y = 1.50661 per B;
- y = 3.992471 per P;
- y = 12.0303 per System.
  Questi valori sono stati calcolati a partire dai valori ottenuti in fase di verifica.
  I grafici convergono ma c'è una leggera differenza tra i valori attesi dal modello analitico

e i risultati delle run. Il server B è quello più vicino al sistema analitico.



8. Design degli esperimenti

Per gli esperimenti si è scelto di utilizzare simulazioni ad orizzonte finito seguendo il metodo del Batch Mean; si effettua una lunga run suddivisa che viene poi suddivisa in k batch contigui con dimensione pari b.
I parametri b e k sono stati individuati tramite programma acs.java fornito insieme al libro di testo. In particolare, sono stati raccolti in un file di testo i tempi di risposta globali di n (n = 1000000) job. Il file di testo poi viene passato in input alla classe Acs che poi calcola l'autocorrelazione dei tempi di risposta dei job con lag differenti.
Come linea guida è stata scelta la seguente: " Banks, Carson, Nelson, and Nicol (2001, page 438) recommend that the batch size be increased until the lag one autocorrelation between batch means is less than 0.2."
I valori ottenuti sono b = 4096 e k = 244, ma il parametro k è sufficiente che sia maggiore di 32, quindi nelle simulazioni k è impostato a 100.

Durante le simulazioni, al completamento di un batch viene calcolata la media delle metriche interessanti X_i. Con tali medie si calcola un intervallo di confidenza:

- $\bar{X} \pm t_{k-1,0.975} \frac{s_X}{\sqrt{k}}$

dove s_X è la deviazione standard tra medie con k-1 gradi di libertà. Livello di confidenza pari a 95%.

8.1 Obiettivo 1

In questa sezione si analizza come variano le metriche del sistema e dei server per differenti parametri di tasso di arrivi. Come nel caso di studi presente sul libro di testo, si effettuano

più simulazioni con il valore di λ che varia da 0.5 req/s fino a 1.2 req/s incrementandolo di 0.05 req/s.

8.2 Obiettivo F2A

In questa sezione si si confrontano le prestazioni del modello F2A con il modello base seguendo la stessa metodologia dell'obiettivo precedente.

8.3 Obiettivo SCALING

In questa sezione si analizza l'impatto dello scaling orizzontale sul Server B e sul sistema. Lo scaling viene influenzato dal parametro C, ovvero il numero di job massimi che può essere presente in una copia del Server B. Quindi in questi esperimenti si vede l'impatto di questo parametro sulle prestazioni del sistema. Successivamente si analizza come il parametro C impatta il numero medio di copie del server B.

9. Analisi risultati

In quest'ultima sezione sono mostrati i risultati degli esperimenti per ogni obiettivo come descritto nella sezione precedente.
Poiché alcuni esperimenti prevedono il variare di parametri (come lambda), nelle tabelle vengono riportati solo alcune batch means. Per avere tutte le batch means di tutti gli esperimenti, è possibile consultare i file csv presenti nella repo Github. Nel file README si può trovare la spiegazione dei vari file csv.

9.1 Obiettivo 1

I grafici sottostanti riportano l'andamento di tempo di risposta medio, popolazione, throughput, utilizzazione e waiting time in base al tasso di arrivi lambda. In questi grafici gli intervalli verticali in rosso rappresentano l'intervallo di confidenza.
Possiamo notare come sia ovviamente il server B a raggiungere valori molto alti per response time medio e popolazione. Il comportamento del sistema, globalmente, è influenzato maggiormente dal server B, tanto da avere curve molto simili.
In questa configurazione, il tempo medio di servizio del server P è pari a 0.4 s, tasso di servizio pari a 2.5 req/s. Dunque, il server P in tutte le metriche assume valori molto più bassi degli altri server.
Con tasso di arrivi pari a 1.2 req/s siamo ancora in condizioni di stabilità. Si ricorda che il tasso di servizi di B è pari a 1.25 req/s; quindi più lambda si avvicina a questi valori, maggiore sarà il tempo di risposta e popolazione.

Tabella per Response Time:

Component	Lambda	Measure	CI
A	0.5	0.3592	$\pm$ 0.0012
B	0.5	1.3328	$\pm$ 0.0068

Component	Lambda	Measure	CI
P	0.5	0.5011	± 0.0015
System	0.5	2.9116	± 0.0081
A	1.2	1.4293	± 0.0262
B	1.2	18.9958	± 1.5940
P	1.2	0.7642	± 0.0041
System	1.2	24.0480	± 1.6288

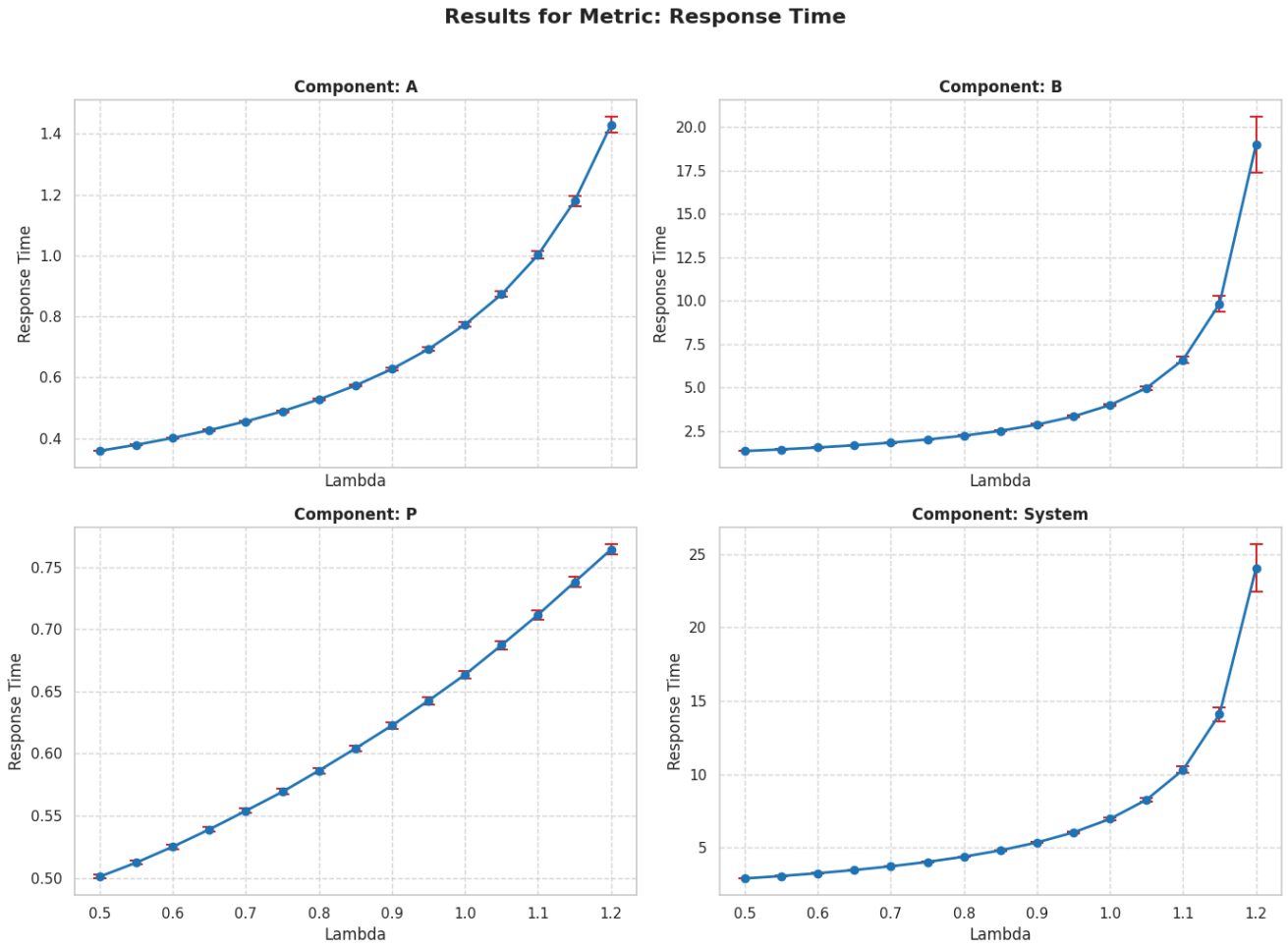
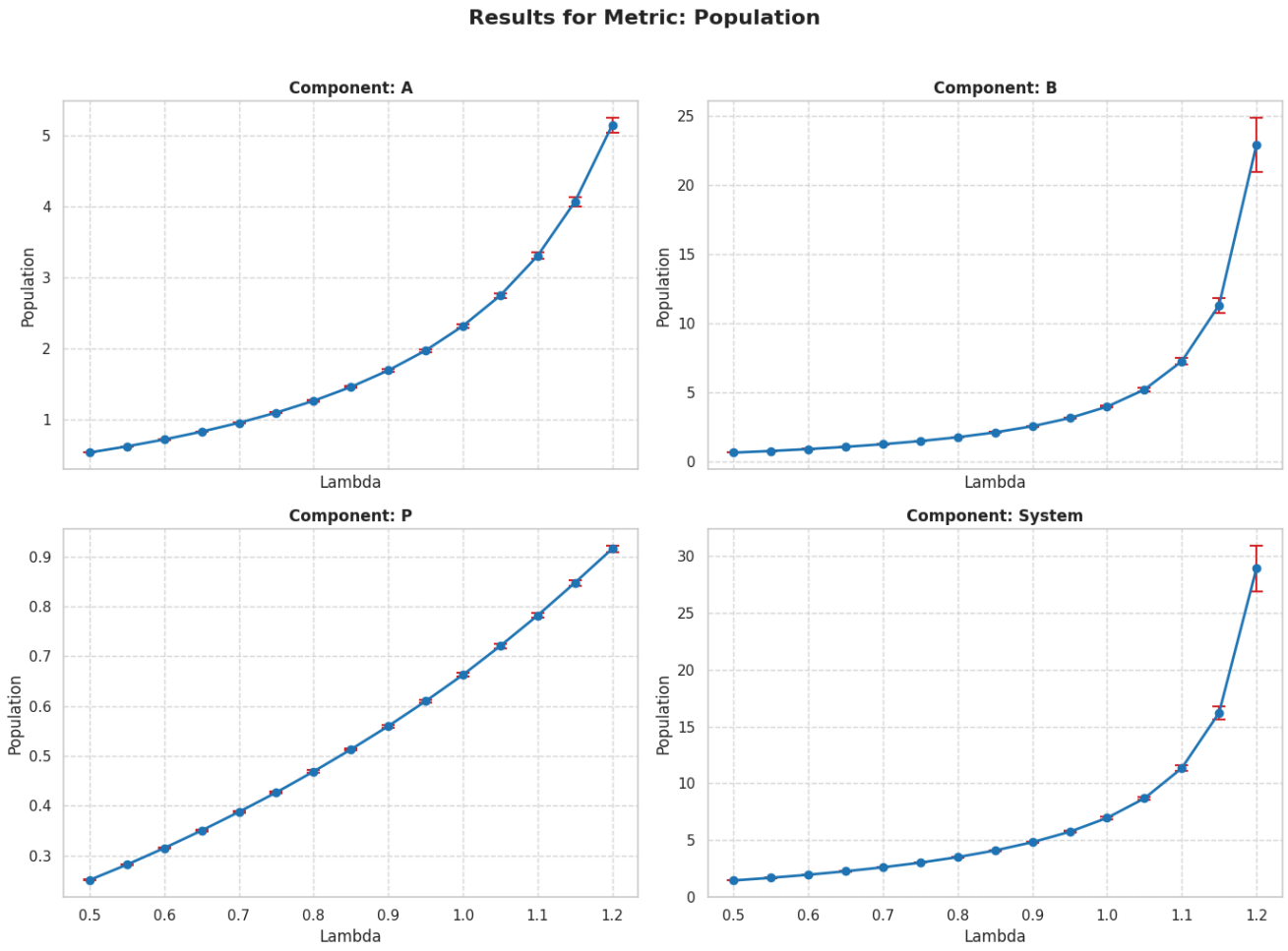


Tabella per Population:

Component	Lambda	Measure	CI
A	0.5	0.5391	± 0.0023
B	0.5	0.6667	± 0.0039
P	0.5	0.2506	± 0.0.001
System	0.5	1.4564	± 0.0057
A	1.2	5.1461	± 0.1015
B	1.2	22.8653	± 1.9504
P	1.2	0.9162	± 0.0059

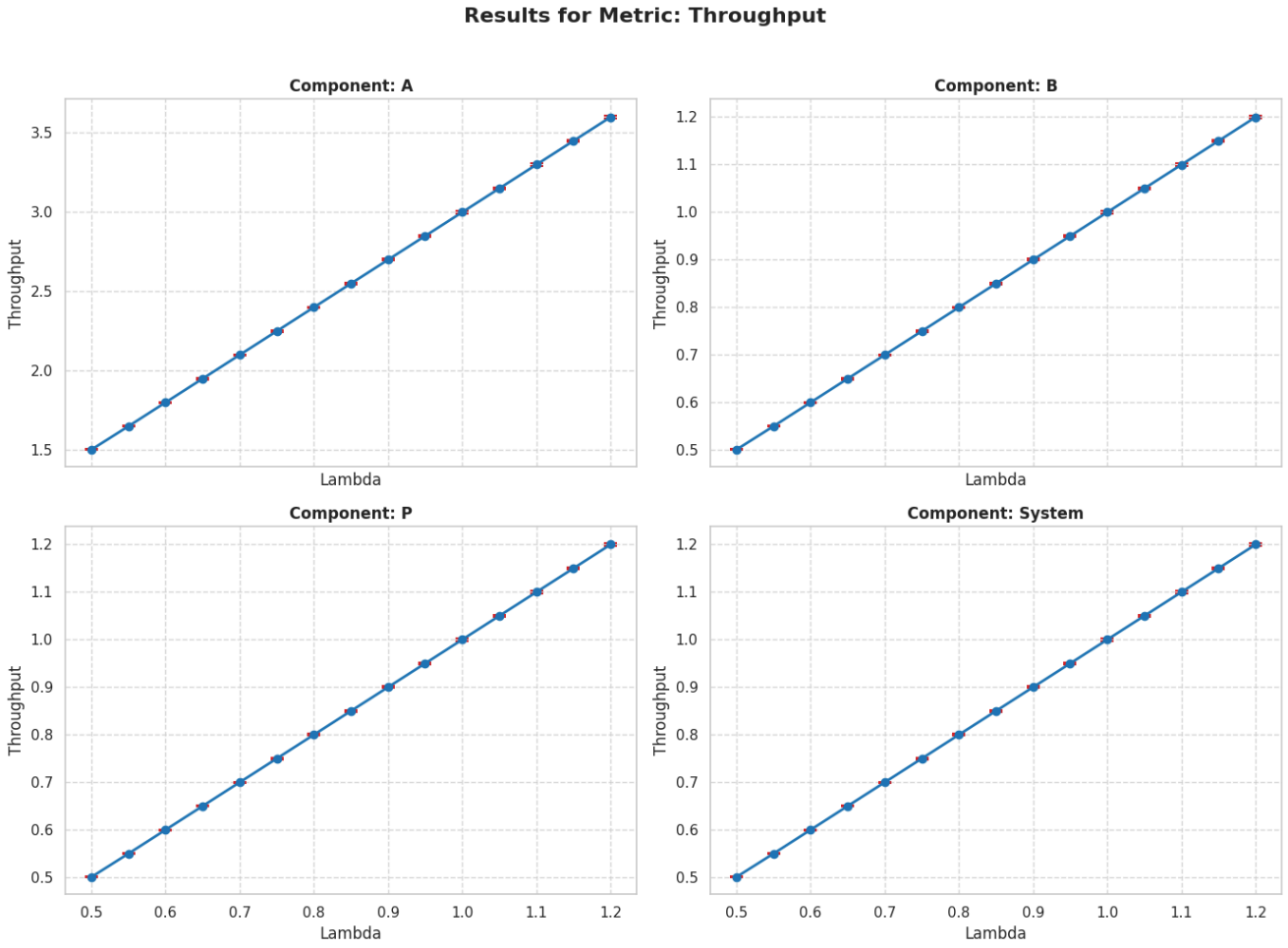
Component	Lambda	Measure	CI
System	1.2	28.9250	± 1.9989



L'andamento del throughput è lineare come atteso, sempre grazie al fatto che con valori di lambda minori di 1.2 req/s, il sistema è in condizione di stabilità. Dunque, il throughput assume proprio il valore del tasso di arrivi lambda.

Tabella per Throughput:

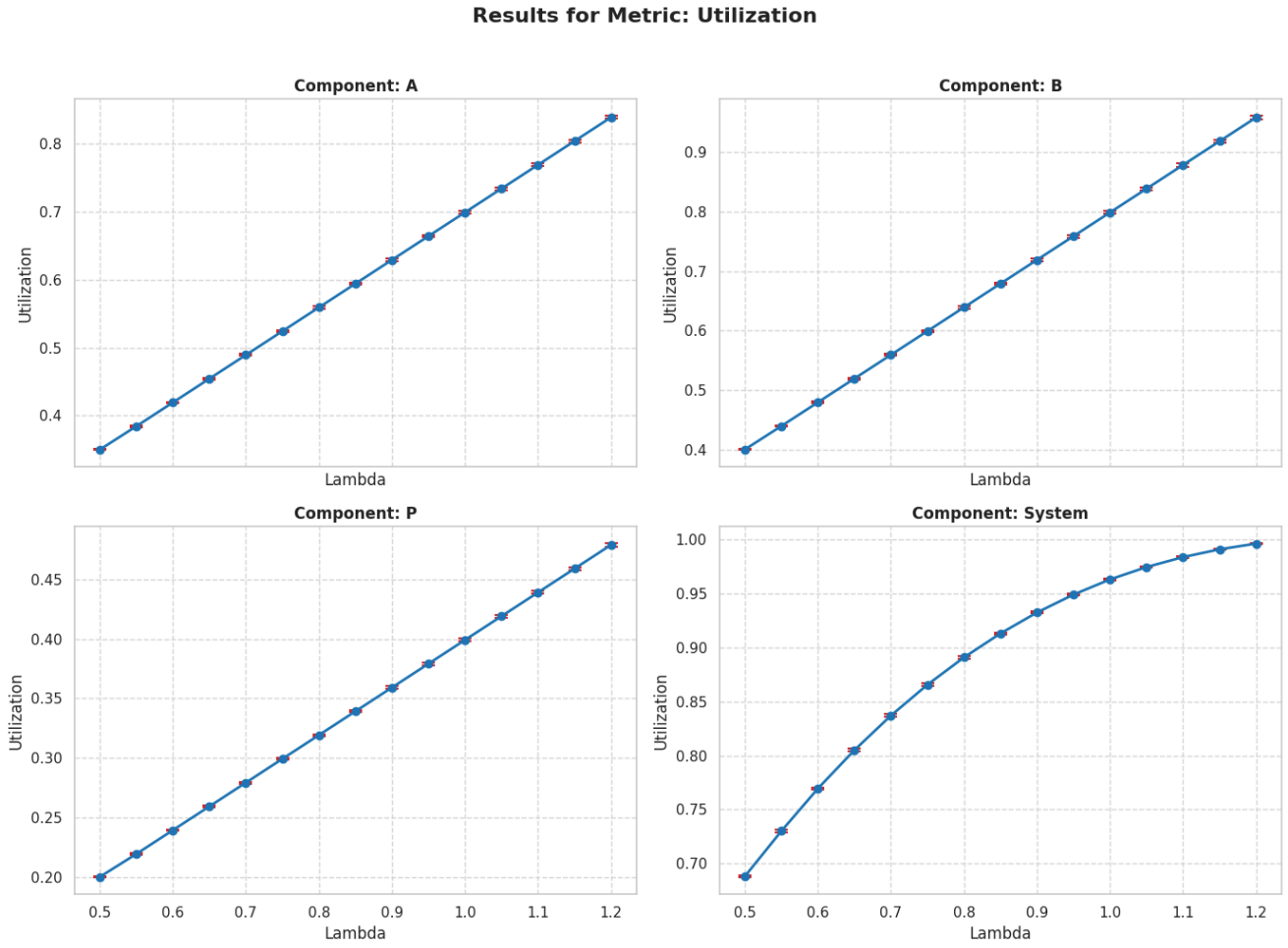
Component	Lambda	Measure	CI
A	0.5	1.5007	± 0.0029
B	0.5	0.5002	± 0.0009
P	0.5	0.5002	± 0.0009
System	0.5	0.5002	± 0.0009
A	1.2	3.5965	± 0.0075
B	1.2	1.1988	± 0.0026
P	1.2	1.1988	± 0.0026
System	1.2	1.1988	± 0.0026



Anche l'utilizzazione per i server assume andamento lineare. Tali andamenti però hanno coefficiente angolare differente, dunque l'utilizzazione del sistema non possiede andamento lineare.
Come nei precedenti grafici, anche qui si può notare come sia il server B ad essere il collo di bottiglia del sistema. Con tasso lambda pari a 1.2 req/s, B raggiunge 0.96 di utilizzazione.
Il server A ha tempo medio di servizio pari a 0.2333 s, tasso di servizio pari a 4.286 req/s, ma ha tasso di arrivi pari a 3.6 req/s. Quindi non raggiunge utilizzazione apri a 0.9 fino a che il tasso di arrivi è pari a 1.286 req/s.

Tabella per Utilization:

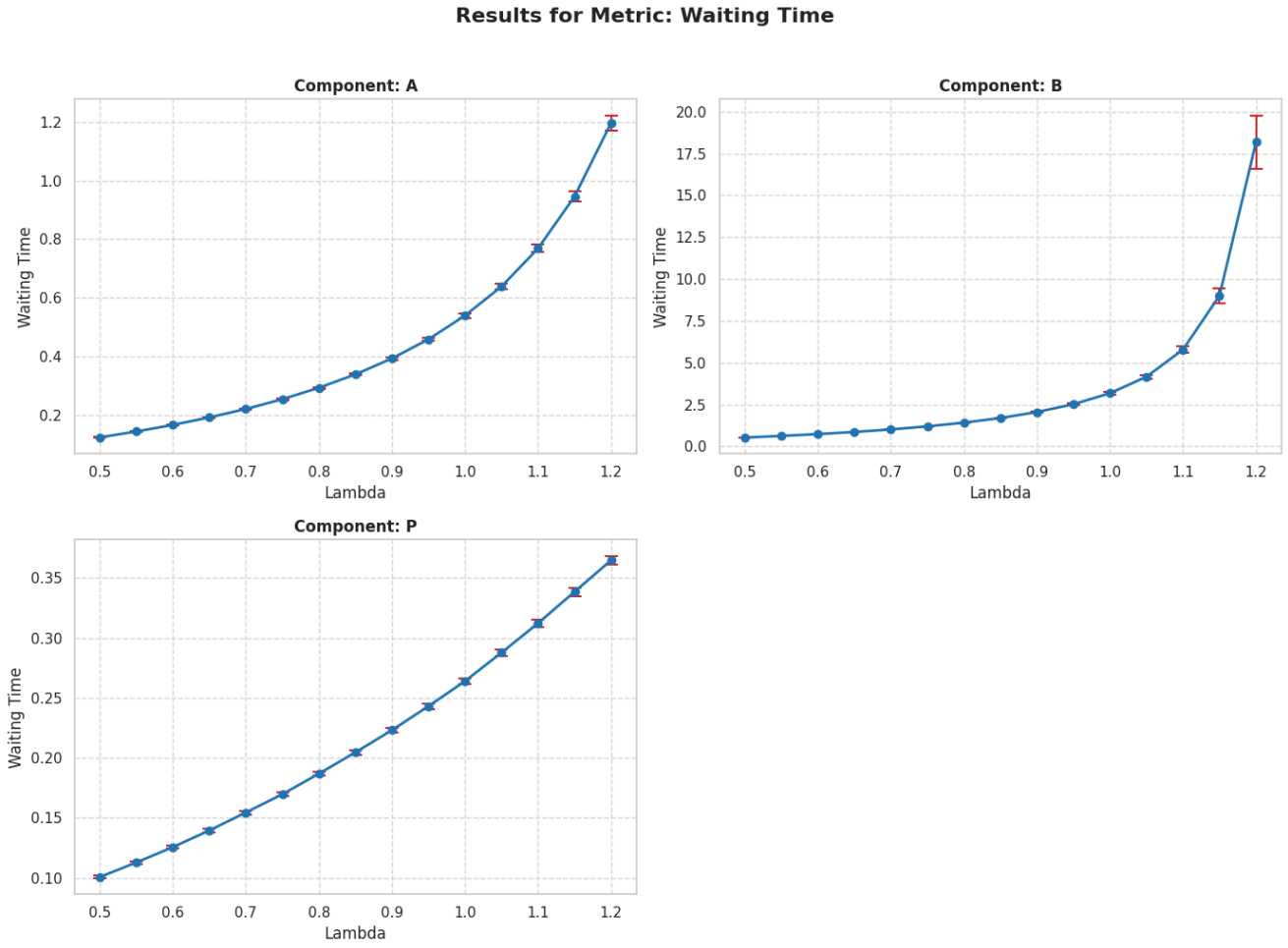
Component	Lambda	Measure	CI
A	0.5	0.3501	± 0.0008
B	0.5	0.3999	± 0.0012
P	0.5	0.2002	± 0.0005
System	0.5	0.6880	± 0.0011
A	1.2	0.8388	± 0.0021
B	1.2	0.9586	± 0.0027
P	1.2	0.4789	± 0.0013
System	1.2	0.9963	± 0.0003



In questi ultimi grafici viene mostrato l'andamento del tempo di attesa. Per il server B possiamo notare incrementi molto ampi da lambda 1.1 req/s in poi che corrispondono proprio a valori di utilizzazione pari maggiori di 0.8.
Poichè si è scelto di non misurare waiting time per il sistema totale, non è presente tale grafico.

Tabella per Waiting Time:

Component	Lambda	Measure	CI
A	0.5	0.1259	± 0.0009
B	0.5	0.5330	± 0.0057
P	0.5	0.1006	± 0.0009
A	1.2	1.1961	± 0.0260
B	1.2	18.1961	± 1.5932
P	1.2	30.3646	± 0.0037



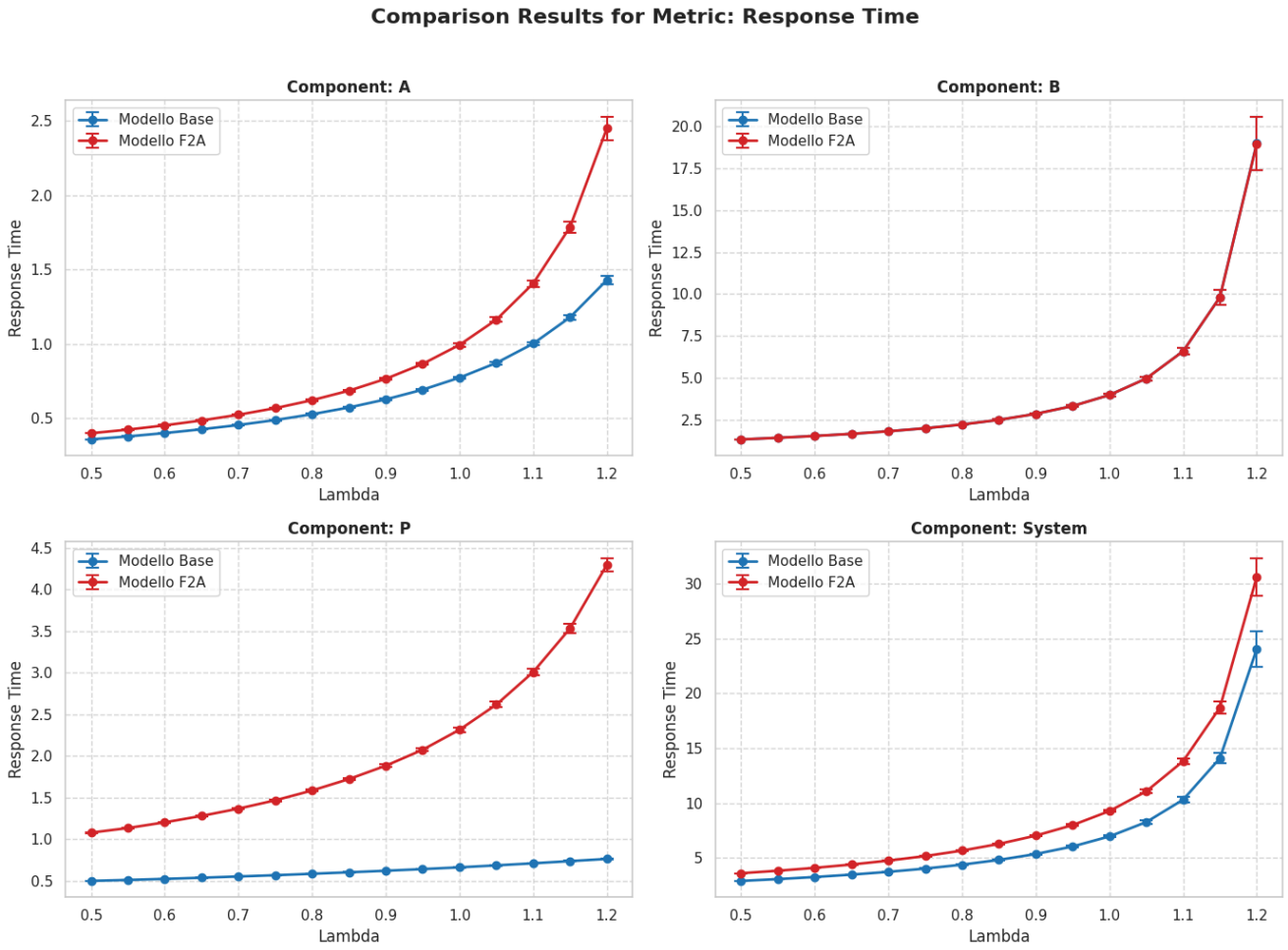
9.2 Modello F2A

Nei seguenti grafici, il modello rosso corrisponde al modello F2A e il modello blu corrisponde al modello base.
Si ricorda che nel modello F2A, la differenza con il modello base consiste nell'aumento del tempo di servizio medio del server P e server A. Tra i due, il maggiore impatto lo subisce il server P, come si può vedere da tutti i grafici, infatti il suo tempo medio di servizio passa da 0.4 s a 0.7 s, molto vicino al tempo medio di servizio del server B.
Il server B non viene alterato, dunque nei grafici per la component B compare solo la curva in rosso, perchè coincide con la curva in blu della sezione precedente.
Il collo di bottiglia rimane sempre il server B, dunque il tempo medio di risposta del sistema aumenta ma non di molto.

Tabella Response Time (curva rossa):

Component	Lambda	Measure	CI
A	0.5	0.4004	± 0.0014
B	0.5	1.3327	± 0.0069
P	0.5	1.0798	± 0.0048
System	0.5	3.6139	± 0.0103
A	1.2	2.4481	± 0.0806

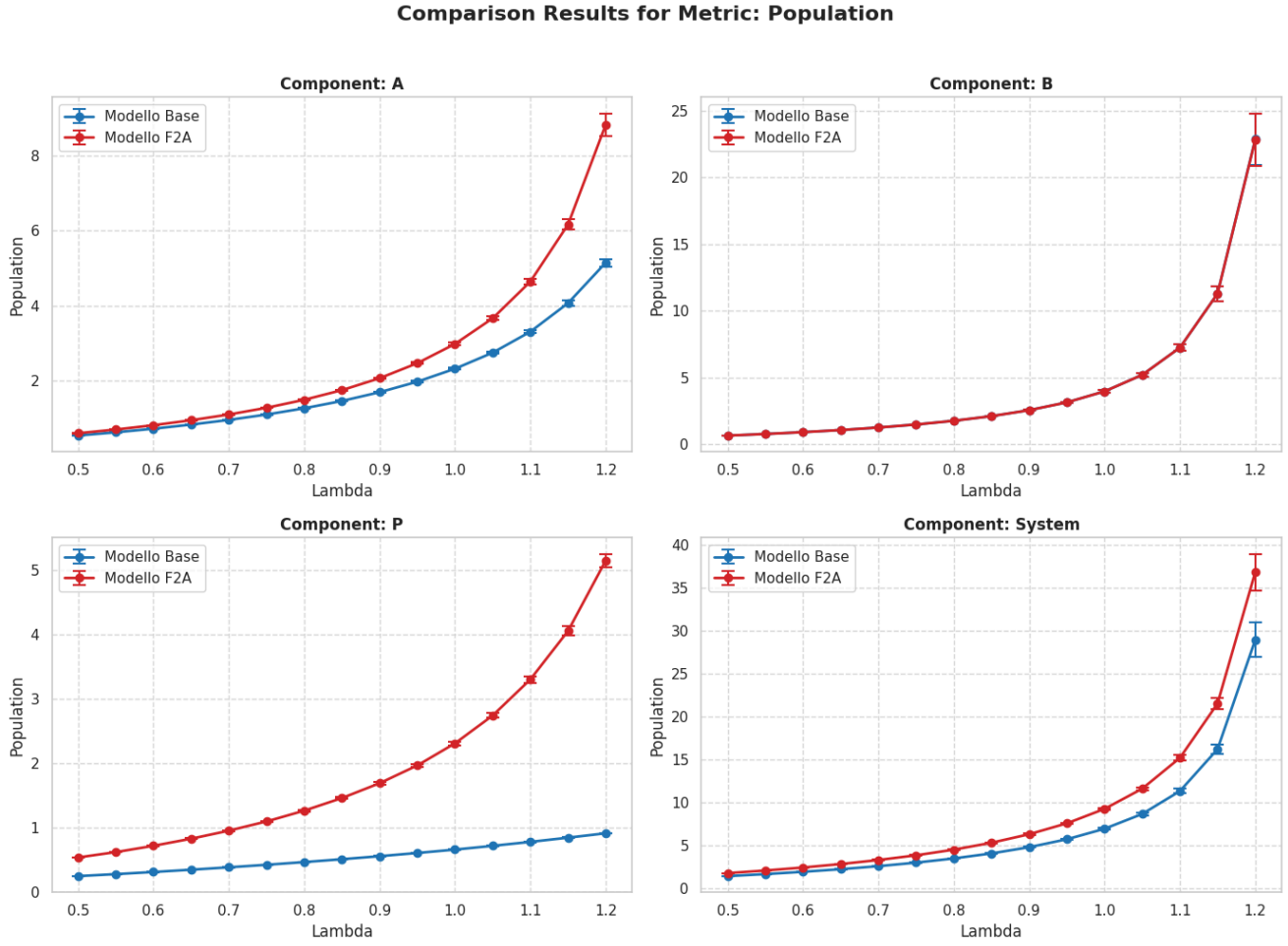
Component	Lambda	Measure	CI
B	1.2	18.969	± 1.579
P	1.2	4.292	± 0.083
System	1.2	30.606	± 1.730



Molto simile è il comportamento dell'utilizzazione.

Tabella Population (solo curva rossa):

Component	Lambda	Measure	CI
A	0.5	0.6009	± 0.0026
B	0.5	0.6667	± 0.0039
P	0.5	0.5401	± 0.0028
System	0.5	1.8078	± 0.0073
A	1.2	8.823	± 0.304
B	1.2	22.831	± 1.931
P	1.2	5.151	± 0.105
System	1.2	36.801	± 2.132



In questa configurazione il server A, con tasso di arrivi pari a 1.2 req/s, raggiunge utilizzazione pari a 0.9. Per avere questo effetto è bastato aumentare di 0.05 s il tempo di servizio medio della classe 2 (server A).
Il server P invece non tocca utilizzazione pari a 0.9 finché il tasso di arrivi non arriva a 1.286 req/s.

Tabella Utilization (solo curva rossa):

Component	Lambda	Measure	CI
A	0.5	0.3751	± 0.0009
B	0.5	0.3999	± 0.0012
P	0.5	0.3504	± 0.0010
System	0.5	0.7563	± 0.0011
A	1.2	0.8987	± 0.0022
B	1.2	0.9586	± 0.0027
P	1.2	0.8381	± 0.0024
System	1.2	0.9992	± 0.00001

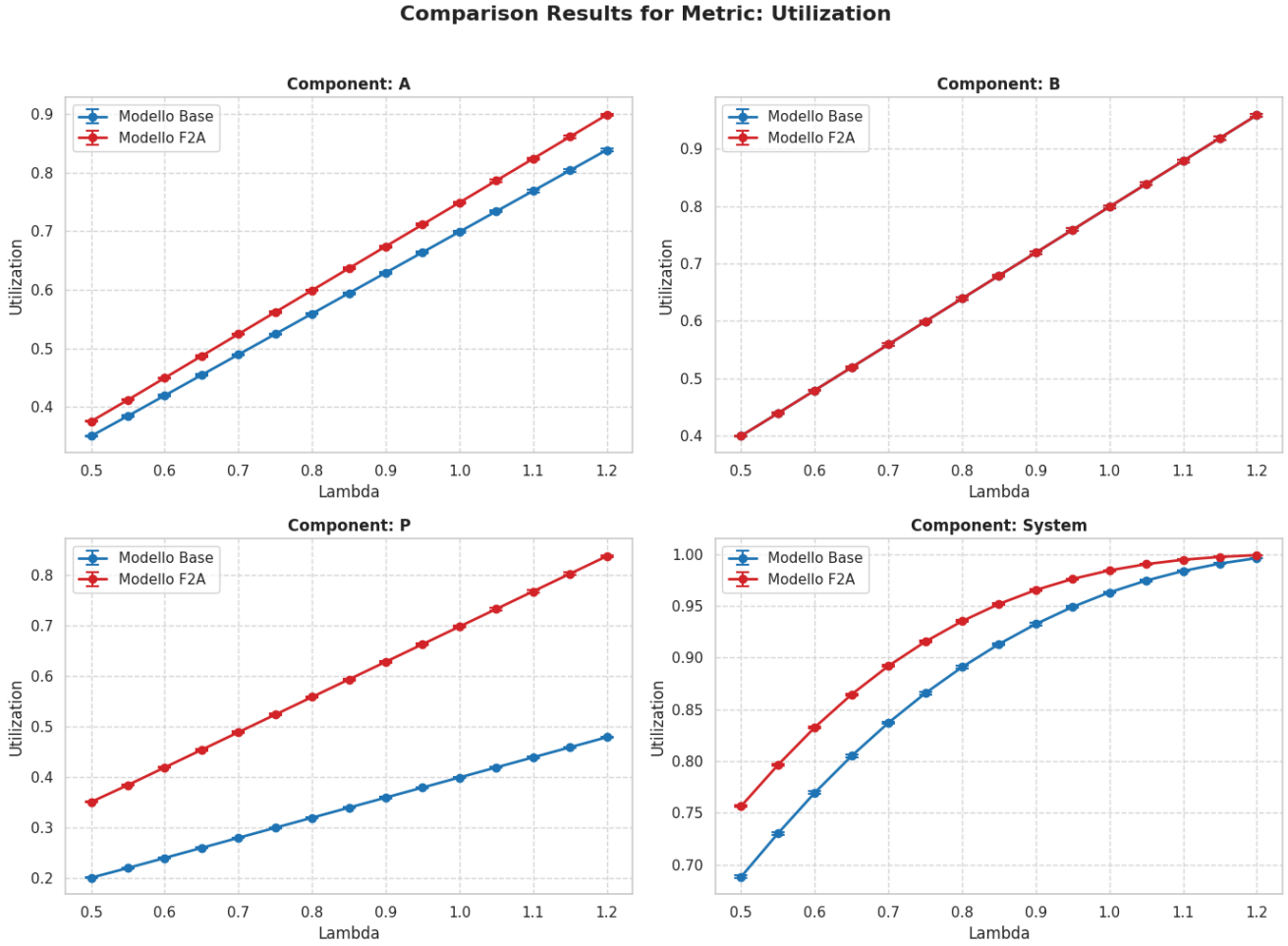
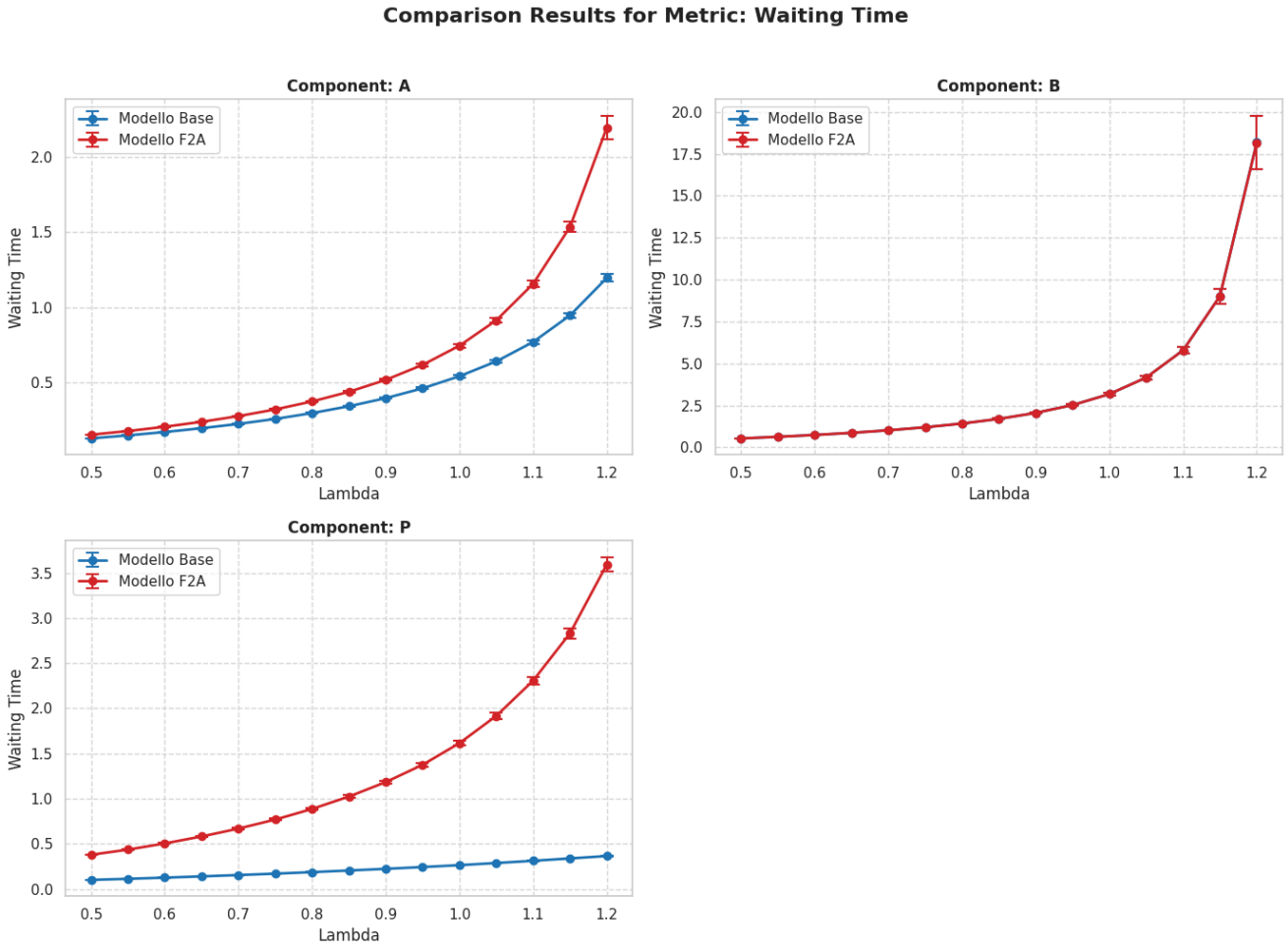


Tabella per Waiting Time (solo curva rossa):

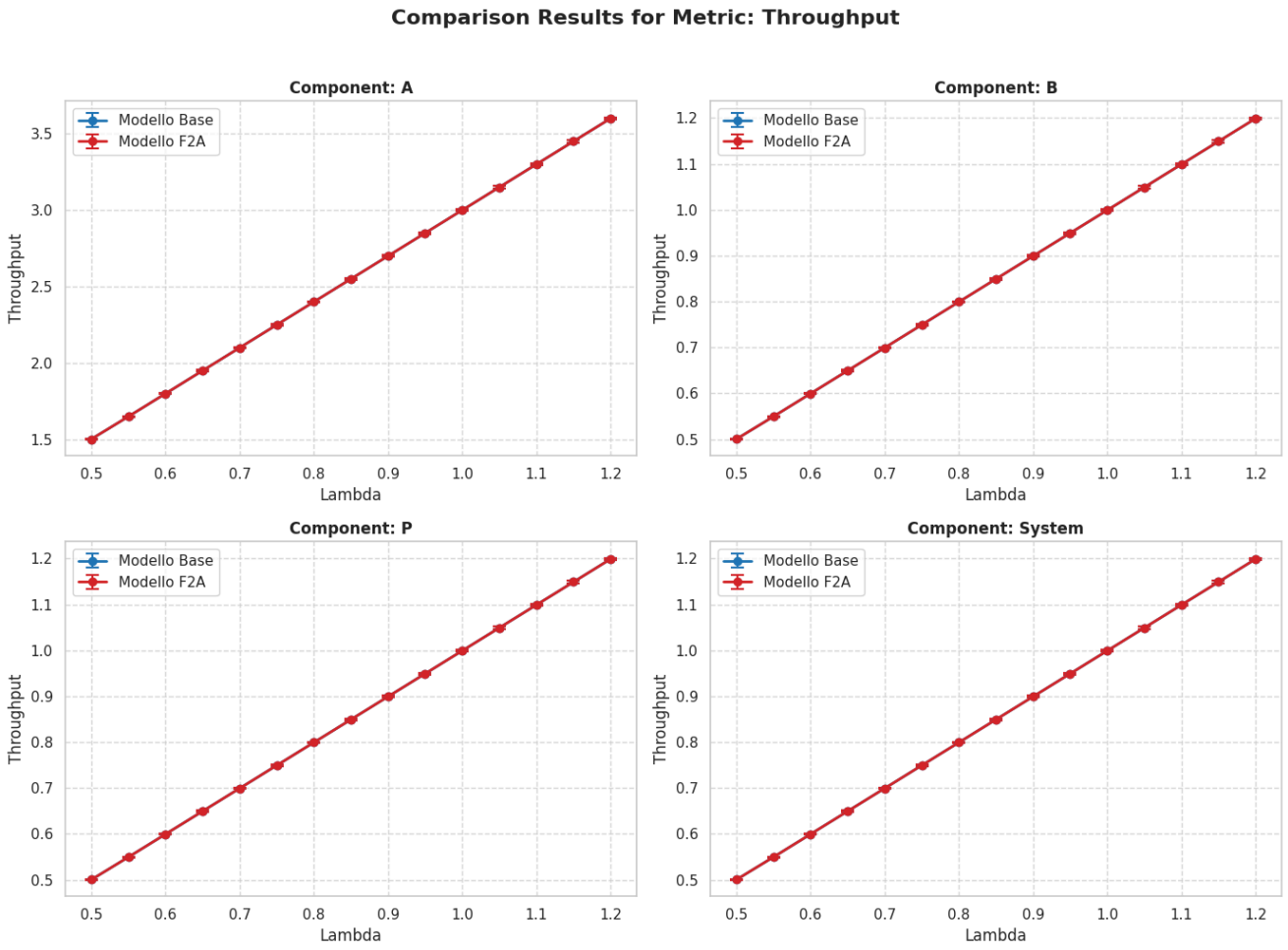
Component	Lambda	Measure	CI
A	0.5	0.1504	± 0.0011
B	0.5	0.5330	± 0.0057
P	0.5	0.3791	± 0.0038
A	1.2	2.198	± 0.0804
B	1.2	18.17	± 1.578
P	1.2	3.593	± 0.0821



Infine, sono riportati risultati per il throughput. Ovviamente il sistema è ancora a condizione di stabilità, dunque anche qui il throughput è dettato dal tasso di arrivi e i valori sono gli stessi del modello base (dunque i due grafici coincidono). I valori in tabella coincidono con quelli del caso base.

Tabella Throughput (solo curva rossa):

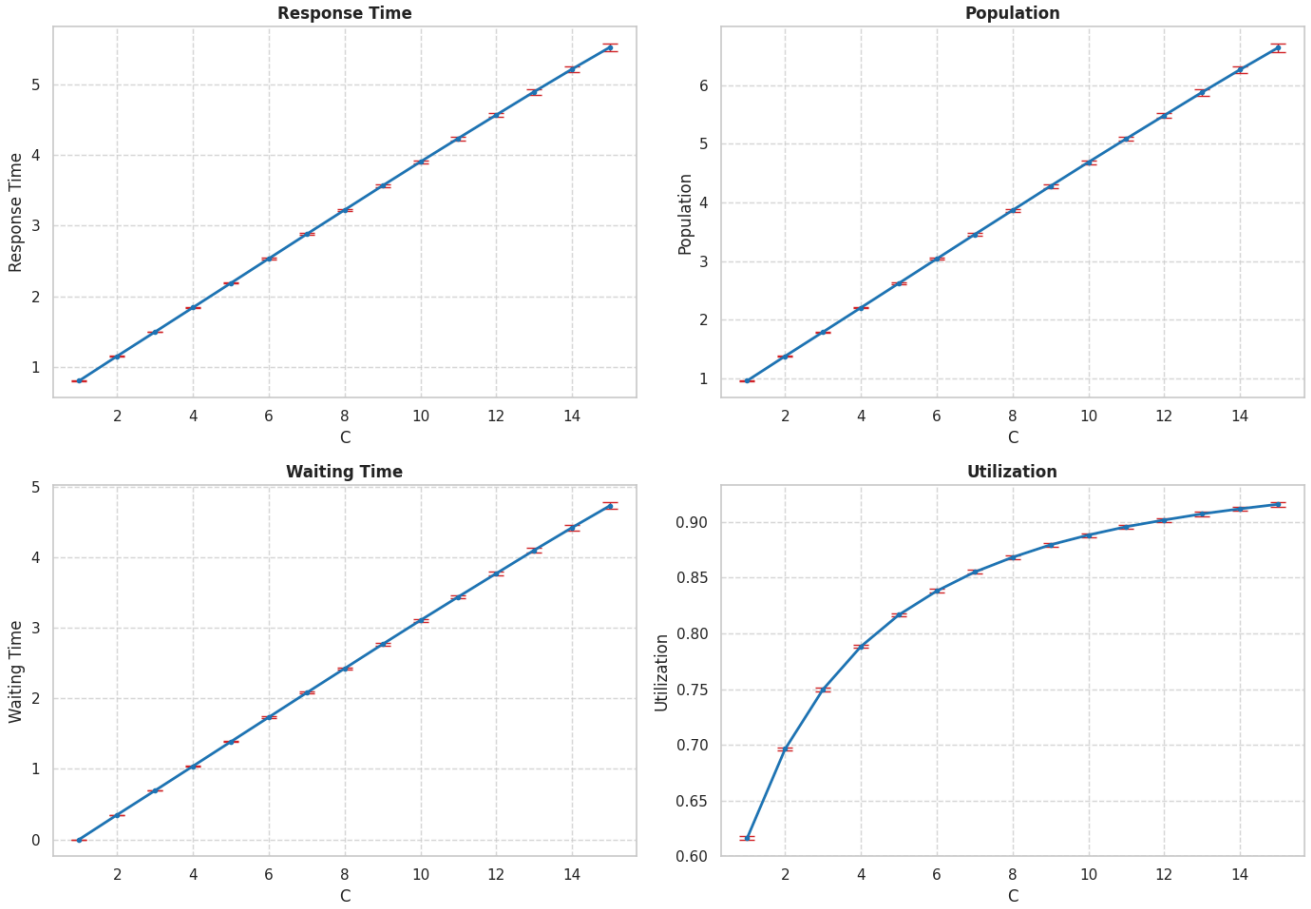
Component	Lambda	Measure	CI
A	0.5	1.5007	± 0.0029
B	0.5	0.5002	± 0.0009
P	0.5	0.5002	± 0.0009
System	0.5	0.5002	± 0.0009
A	1.2	3.5965	± 0.0076
B	1.2	1.1988	± 0.0026
P	1.2	1.1988	± 0.0026
System	1.2	1.1988	± 0.0026



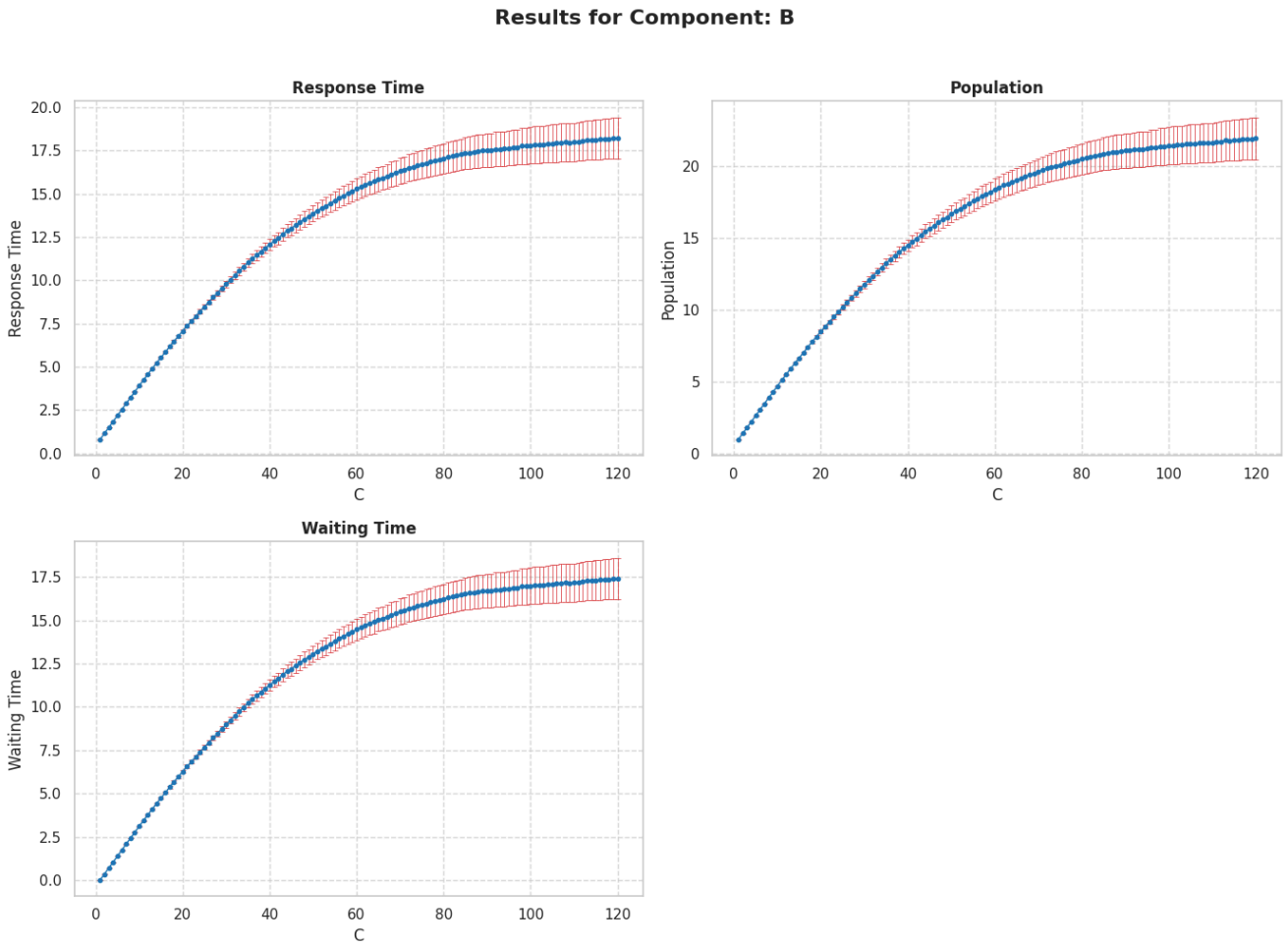
9.3 Modello SCALING

Nei primi grafici si può osservare il comportamento dei server e del sistema dove Server B ha la capacità di replicarsi. Il parametro C dei grafici (asse x) rappresenta il numero di job massimi che una copia del server B può possedere.
Ovviamente questo cambiamento impatta solo il server B e le prestazioni globali; quindi, Server A e Server P non risentono di cambiamenti.
Ovviamente, fissare valori molto bassi per il parametro C diminuisce di molto il tempo medio di risposta, popolazione media e tempo medio di wait del server B.

Per il primo grafico, il parametro C varia da 1 a 15. Si può osservare come con C = 1, non esiste waiting time e il tempo di risposta medio coincide con il tempo medio di servizio del server B, infatti con C = 1 possiamo vedere questo server come un Infinite Server.



In questo secondo grafico il parametro varia da 1 a 120. Ovviamente con $C \to \infty$ abbiamo che il server B si comporta proprio come nel caso senza scaling.

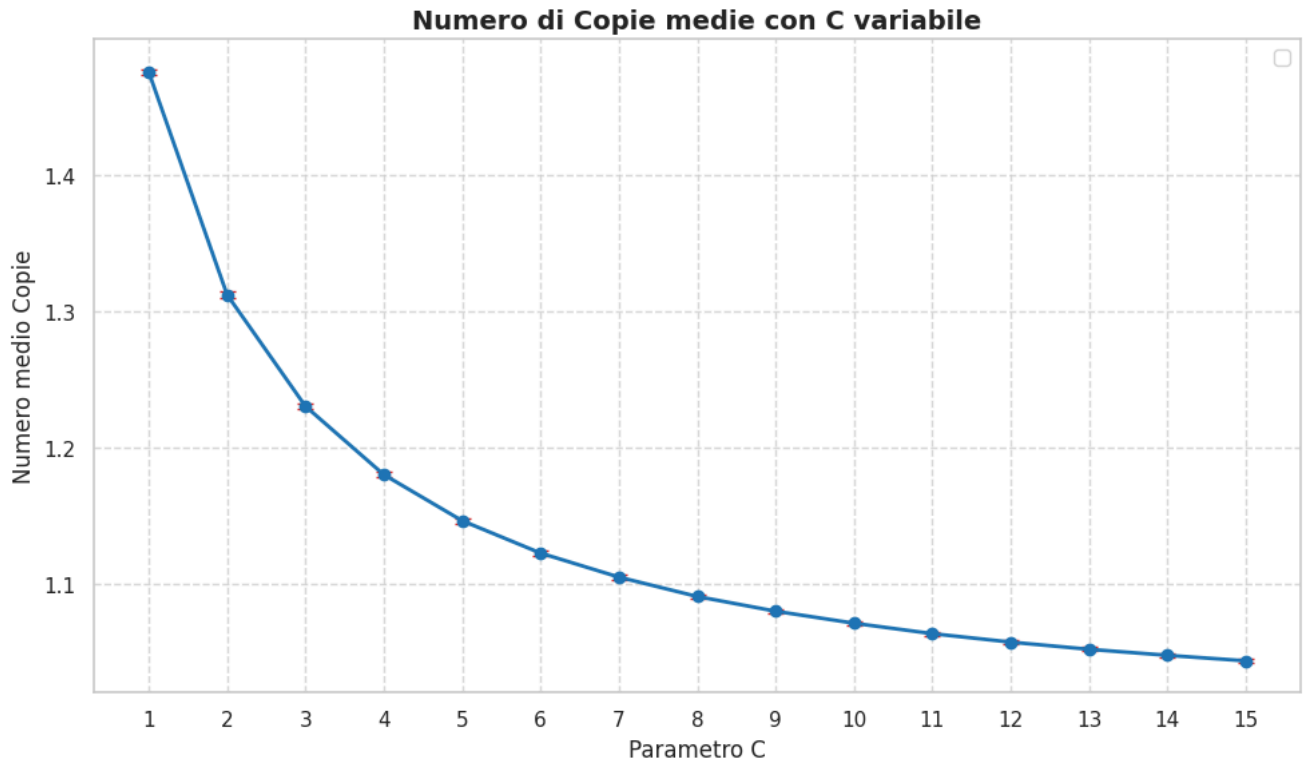


Ha senso osservare il numero di copie medie del server B al variare del parametro C. Attenzione, nel modello computazionale, se una copia di B si trova senza job, allora genera evento DESTROY che viene eseguito immediatamente. Nel codice è possibile cambiare questo comportamento inserendo un delay per fare in modo che, se entro questo delay arriva un job alla copia, questa non viene distrutta e l'evento DESTROY viene eliminato. Se si inserisce questo delay, il numero medio di copie aumenta, ma non è più significativo poichè ci possono essere copie vive ma senza job.

Qui sotto sono riportati sia i risultati in forma tabellare sia come grafico.

Parametro C	Numero Copie Medio	CI
1	1.4752	± 0.0022
2	1.3118	± 0.0020
3	1.2300	± 0.0019
4	1.1798	± 0.0018
5	1.1460	± 0.0017
6	1.1223	± 0.0017
7	1.1047	± 0.0016
8	1.0904	± 0.0016
9	1.0797	± 0.0016
10	1.0709	± 0.0015
11	1.0632	± 0.0015
12	1.0570	± 0.0015
13	1.0517	± 0.0015
14	1.0473	± 0.0014
15	1.0433	± 0.0014

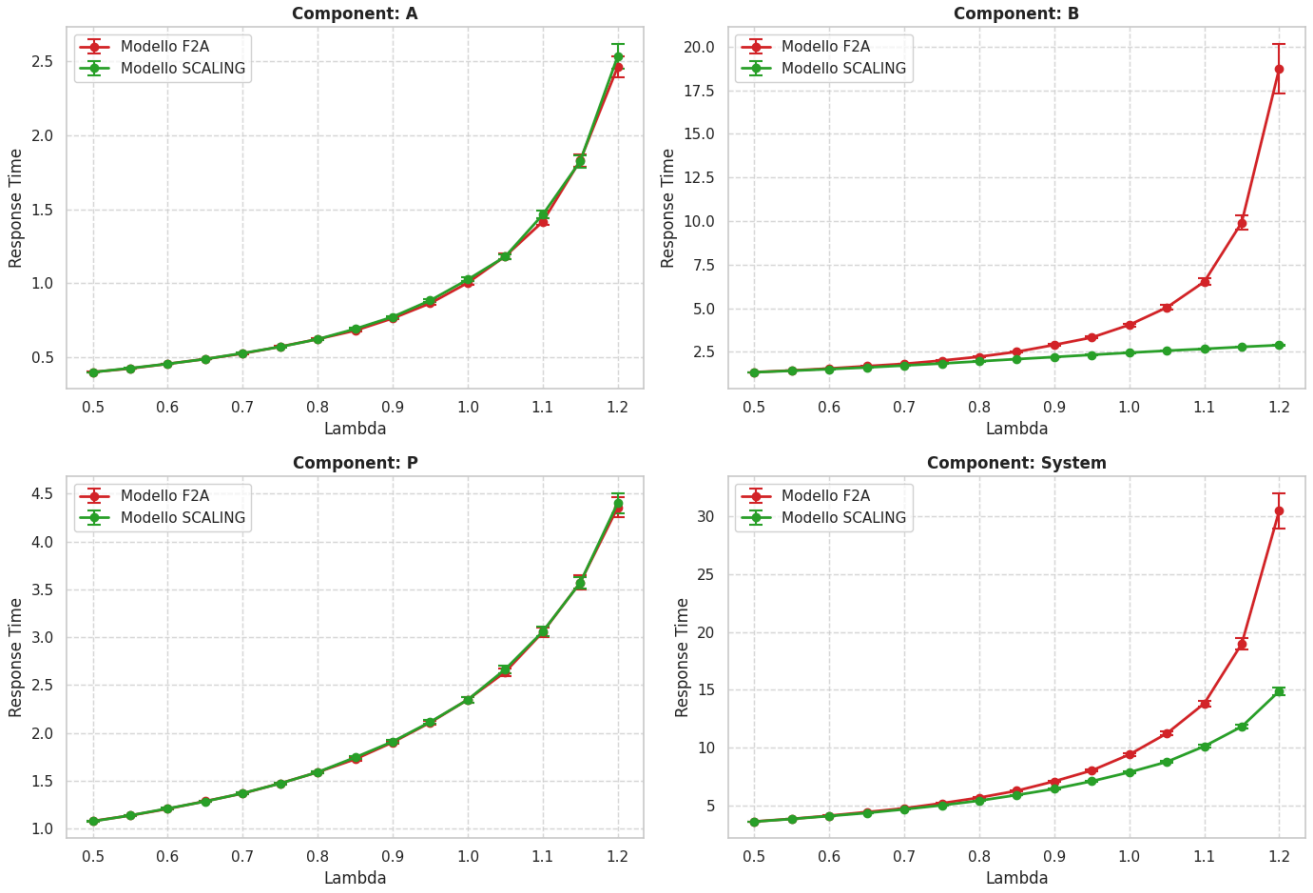
Grafico che mostra il numero medio di copie del server B rispetto al numero di job massimi all'interno delle copie.



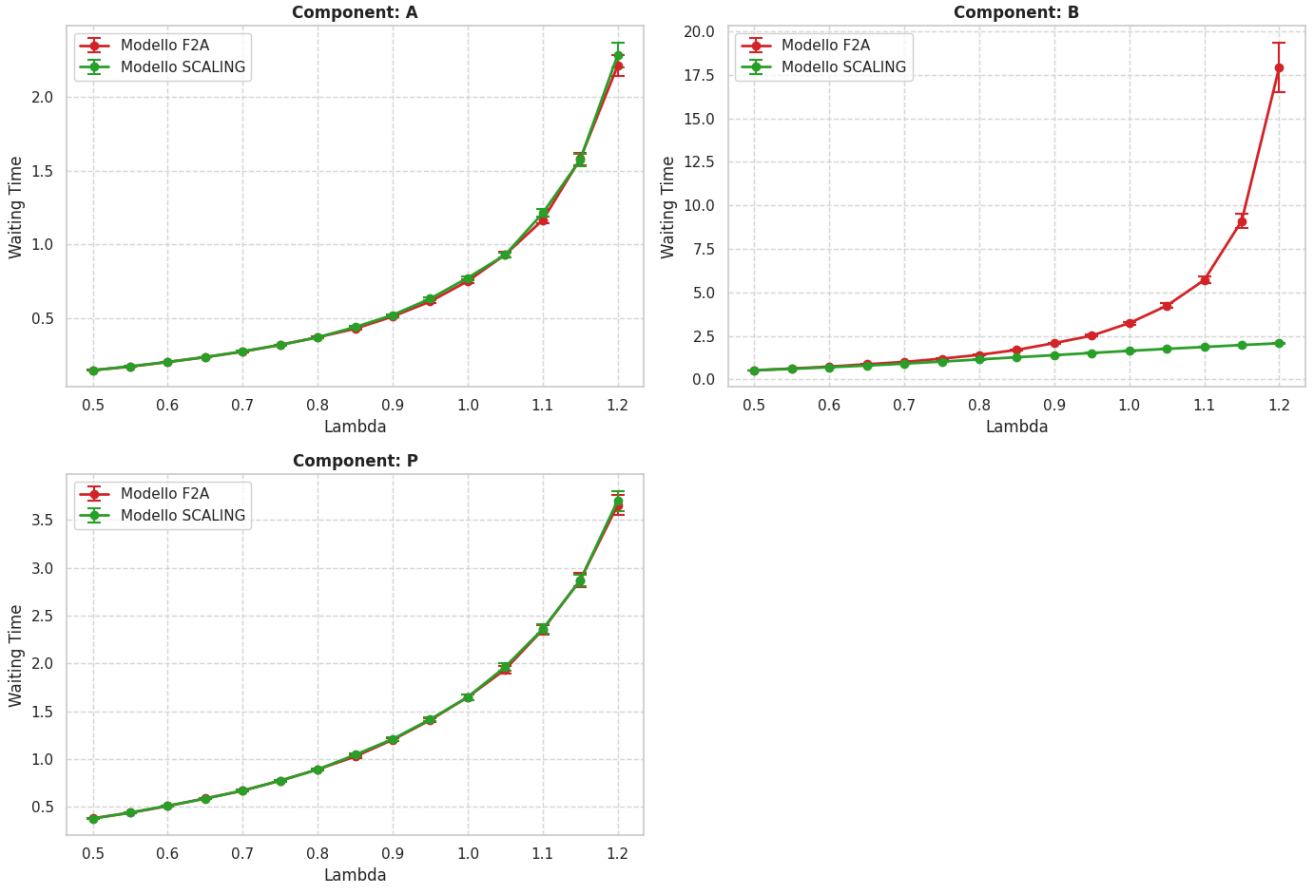
Infine si fissa C = 9 e si confronta il suo comportamento con F2A con λ variabile.
Si nota subito che tempi medi di risposta e wait crollano significativamente per il server B.
Anche la popolazione diminuisce di molto.
Si riporta in tabella i valori di response time medio e popolazione media a confronto per il caso $\lambda = 1.2$.

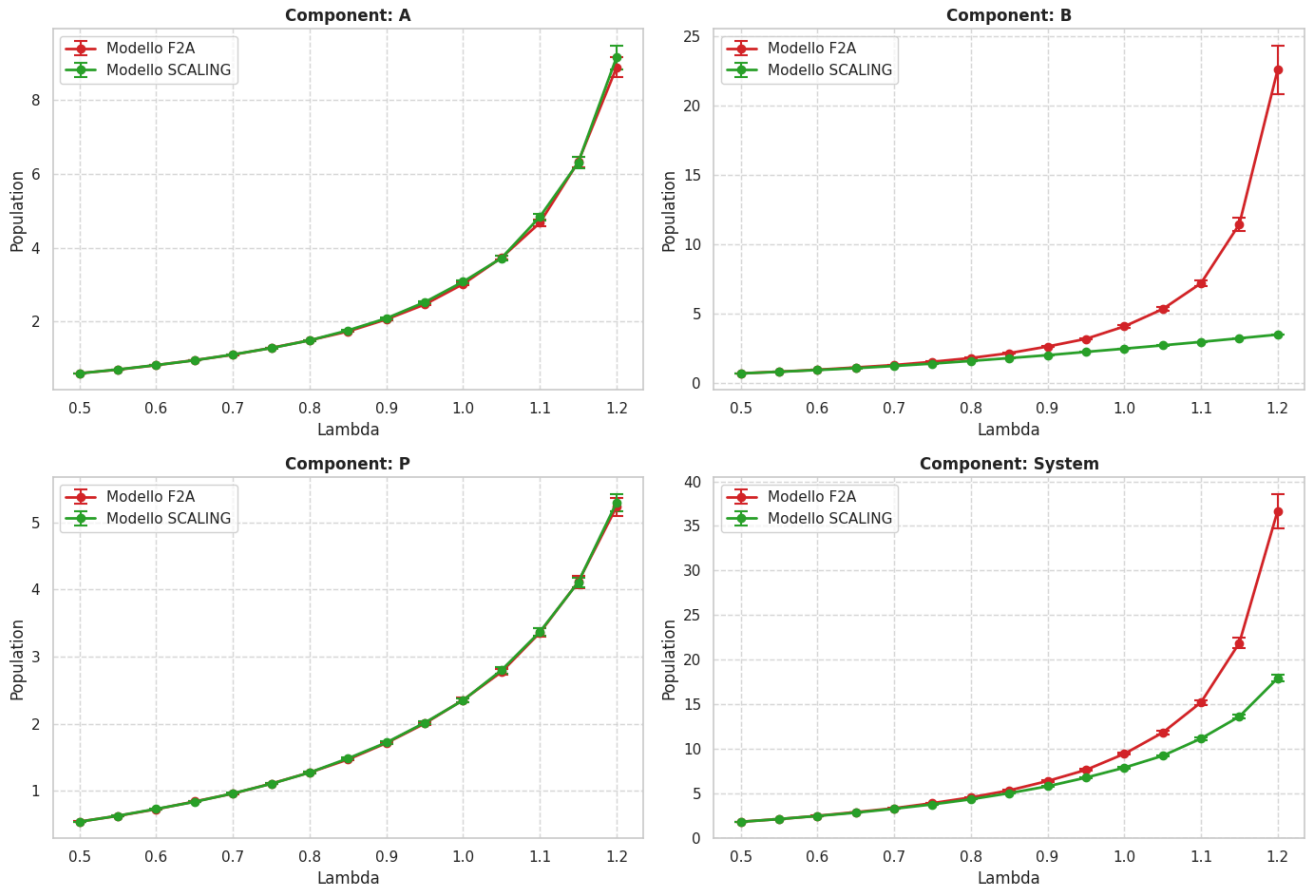
Model	Metric	Component	Value	CI
SCALING	Response Time	A	2.5321	0.0859
F2A	Response Time	A	2.4481	0.0806
SCALING	Response Time	B	2.8813	0.0136
F2A	Response Time	B	18.9694	1.5792
SCALING	Response Time	P	4.4011	0.1049
F2A	Response Time	P	4.2925	0.0828
SCALING	Response Time	System	14.8749	0.3069
F2A	Response Time	System	30.6066	1.7298
SCALING	Population	A	9.1544	0.3249
F2A	Population	A	8.8225	0.3044
SCALING	Population	B	3.4645	0.0201
F2A	Population	B	22.8315	1.9315
SCALING	Population	P	5.2969	0.1325
F2A	Population	P	5.1507	0.1055

Comparison Results for Metric: Response Time



Comparison Results for Metric: Waiting Time





10 Conclusioni

Il modello SCALING è una modifica molto semplice del caso base. Apporta dei miglioramenti al sistema ma poco significativi rispetto a scelte di modifiche del sistema reale.
Inoltre. dagli esperimenti si evince che non c'è un parametro C (numero di job massimi per copia) che sia migliore di altri. Anche le assunzioni molto semplici del caso di studi non permettono di definire dei parametri ottimali.

Dunque, la scelta di un valore per il parametro C dovrebbe essere guidato da altre dinamiche del sistema, come costo delle repliche dei server, tempo di attivazione e costo di mantenimento.