UNIVERSITA' degli STUDI di ROMA
TOR VERGATA

# Un esempio di start-up

PMCSN 2020/2021

Moroni Valerio, Perrone Valentino

# Introduzione

Il caso di studio che abbiamo affrontato è ispirato al mondo del cloud computing, in particolare abbiamo provato a calarci nei panni di una nuova start-up di un social network che, avendo risorse fisiche limitate (non dispone di un intero data center, ma si appoggia ad un numero limitato di server) decide di affidarsi al cloud per gestire i picchi di carico, quando necessario. L'elasticità, cioè  il potenziale con cui un sistema è in grado di adattarsi ai cambiamenti del carico di lavoro attraverso il provisioning e il de-provisioning delle risorse autonomamente, in modo tale che in ogni momento le risorse disponibili corrispondano il più possibile alla domanda corrente, rende il cloud estremamente versatile ed un "must" per le nuove aziende che hanno intenzione di crescere ed espandersi su larga scala. Questo perché si deve fornire sin da subito un servizio che garantisca il minor periodo di downtime possibile, una miglior distribuzione del carico e, soprattutto, una riduzione dei tempi d'attesa per il servizio offerto, il tutto cercando di ammortizzare le spese iniziali.

# Descrizione del sistema e modello concettuale

Il sistema analizzato si compone di due entità principali: un insieme di 10 server, che possono essere ricondotti al modello multi-server, ed il Cloud, nel nostro caso Amazon EC2 (Elastic Compute Cloud) rappresentabile come una serie infinita di macchine virtuali indipendenti tra loro, ciascuna delle quali assume il comportamento di un single-server, istanziato nei momenti di bisogno, on-demand.

Abbiamo suddiviso le richieste in entrata nel sistema secondo due "task": le richieste di lettura (task 1), più rapide da eseguire e le richieste di scrittura (task 2), leggermente più gravose a

livello computazionale. Suddividere le richieste in arrivo in questo modo ci è sembrato particolarmente adeguato al contesto: nel mondo social, con le dovute approssimazioni, ogni richiesta può essere generalizzata come appartenente a queste due categorie. Quando si richiede una lettura, infatti, si sta cercando di visualizzare un contenuto; quest'ultimo può essere un post, un'immagine o un video, una storia, un hashtag, un profilo o una pagina, le statistiche del proprio profilo, il numero di persone con cui si è entrati in contatto e così via. Per la natura di questo tipo di task ci è apparso evidente come, a livello computazionale, risulti meno gravoso recuperare un'informazione in un sistema distribuito e visualizzarla sul dispositivo utilizzato dall'utente rispetto ad un task di tipo 2, quindi una scrittura, che coinvolge anche il salvataggio su un sistema di storage distribuito dell'informazione inserita dallo user. In un social network le scritture sono tutte le richieste che tendono a modificare lo stato del nostro profilo o di quello del sistema complessivo: l'aggiunta di un post, di un'immagine, di un tag, di un commento, di una storia, ma anche l'aggiunta di un mi piace, di una localizzazione per l'aggiornamento di uno status e così via. La generalizzazione delle richieste a questi due tipi di job ci è sembrata, dunque, pertinente al contesto ed al sistema preso in esame.

A coordinare l'interazione tra i due nodi vi è un controllore, situato nella server farm, che gestisce le richieste e le smista verso il nodo più opportuno. Se le richieste dovessero eccedere la capacità massima dei server iniziali, quindi in caso di picco, il controllore si occuperà dinamicamente di allocare nel cloud il numero di macchine virtuali necessarie per soddisfare interamente le richieste e smaltirle. Ciò significa che, per ogni nuova richiesta, il controllore verifica se ogni server della server farm risulti occupato; una volta occupati tutti i server del pool, il controllore istanzia nel Cloud dinamicamente, tramite i servizi AWS EC2, tante macchine virtuali quante sono le richieste in eccesso che il pool di server,

da solo, non riuscirebbe a smaltire. Il task, una volta assegnato, attende il suo tempo di completamento ed esce dal sistema. Da qui si capisce come i servizi di AWS Cloud entrino in gioco solo in caso di saturazione del primo nodo, ossia la server farm con i suoi 10 server. In questo modo si evita di avere perdita nel sistema, garantendo in ogni momento che le richieste in arrivo possano essere soddisfatte.

Questa caratteristica è, di fatto, fondamentale per una start-up che vuole farsi conoscere sin da subito nel mercato dei social network: minore è il downtime dell'app, maggiore è l'affidabilità e, di conseguenza, il bacino d'utenza può ampliarsi efficacemente sin dalle prime battute. Un'app che non può garantire il soddisfacimento di tutte le richieste in arrivo potrebbe risultare inaffidabile e lenta e, di conseguenza, far crollare drasticamente la sua clientela che, affinché lo sviluppo sia costante e continuo, deve risultare soddisfatta già dalle prime esecuzioni.

Il processo d'esecuzione si può schematicamente riassumere in due algoritmi: il primo è quello che si è cercato brevemente di descrivere nelle righe precedenti, il secondo è un miglioramento del primo per poter aumentare le prestazioni del sistema complessivo, tramite un meccanismo di controllo più "intelligente". In quest'ultimo lo scopo è quello di suddividere gli arrivi in maniera ottimizzata: essendo le macchine virtuali di EC2 "infinite" (quindi sempre disponibili), si cerca di inoltrare la maggior parte del lavoro a queste ultime, lasciando i compiti meno gravosi alla server farm. Per ogni nuovo arrivo, infatti, quest'ultima prenderà in carico le richieste se e solo se almeno una delle seguenti condizioni viene soddisfatta: il task in arrivo è una lettura (quindi è di tipo 1), oppure se il numero di task in arrivo alla server farm non ecceda una quantità prestabilita, minore della capacità stessa del pool di server disponibili, nel nostro caso fissata a 8 e la cui motivazione risulterà più chiara nei paragrafi successivi. Se tali

controlli non dovessero essere verificati, il task viene inoltrato ad AWS Cloud.

Come si può evincere anche dagli schemi sottostanti, il modello generale è riconducibile ad un sistema aperto aciclico (questo per assenza di feedback). Il flusso di richieste in entrata è pari al flusso in uscita, senza perdita, dato che ogni task non processabile dalla server farm, viene sempre preso in carico da istanze di macchine virtuali EC2. Infine, ogni task corrispondente ad una richiesta di un utente, nel caso venga servito (e, quindi, la sua richiesta esce dal sistema) e poi voglia nuovamente tornare ad usufruire del servizio, produce nuovamente un task (genera un nuovo arrivo) che entra nel sistema come fosse la prima volta, indipendentemente dal fatto di aver richiesto servizio di recente.

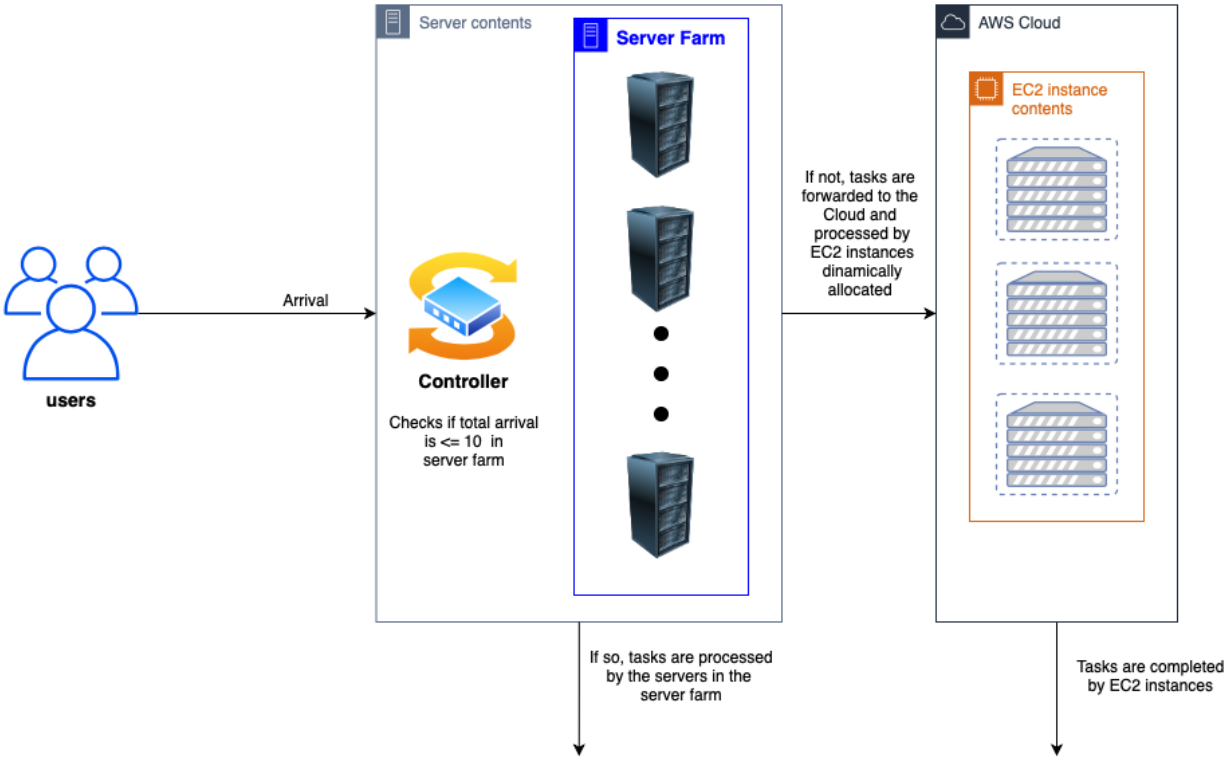



Diagramma raffigurante l'esecuzione del primo algoritmo.

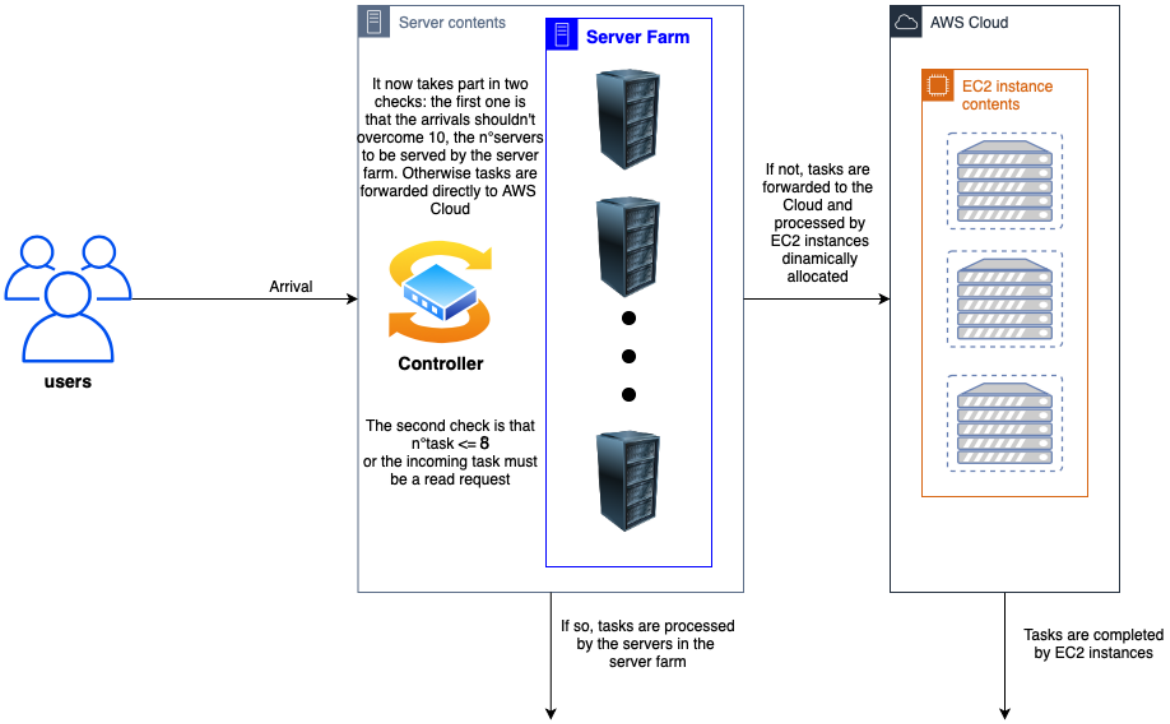



Diagramma raffigurante l'esecuzione del secondo algoritmo.

# Goal ed obiettivi

L'idea di calarsi nei panni di una start-up volta ad introdurre nel mercato un nuovo social network è un argomento pesantemente attuale e di grande interesse. Provare a simularne un ipotetico comportamento, ovviamente con tutte le approssimazioni effettuate per renderlo "studiabile", è stato l'obiettivo principale di tutto il progetto. Una volta che si è riusciti a modellare il sistema, il target finale è stato quello di provare a migliorarne le prestazioni (in particolare il tempo medio di risposta globale per le letture) introducendo un algoritmo migliorativo. Le letture rappresentano, infatti, tra le operazioni più richieste dagli utenti di social network: visualizzare contenuti come post, immagini e video, storie, hashtag, profili o pagine, rientrano in questo caso ed hanno bisogno della minore attesa possibile, per rendere il social più fruibile ed evitare percezioni di caricamenti eccessivi.

# Modello delle specifiche

Per procurarci dati di input necessari per la simulazione abbiamo ricercato principalmente documentazioni di implementazioni di server farm ed istanze di macchine virtuali EC2, analizzando le stime di traffico e di servizio. In caso di assenza, alcuni dei dati da fornire in input sono stati assunti utilizzando conoscenze pregresse o direttamente fornite durante il corso da esempi ed esercizi. Il nostro modello prende in considerazione il fatto di essere una start-up di un social network agli albori: nonostante il numero di server a disposizione non sia trascurabile, la potenza computazionale di ciascuno di essi non è ancora all'altezza della concorrenza, le richieste in arrivo sono ancora piuttosto basse perché la diffusione della piattaforma non è ancora elevata ed i tassi di servizio non sono eccessivi proprio perché ci si immagina di non disporre ancora di un budget sufficiente per poter mantenere un'infrastruttura più prestante. In poche parole, se la nostra start-up si appoggiasse alla sola server farm a disposizione, non riuscirebbe mai a soddisfare interamente il numero di richieste in arrivo ed i server avrebbero in ogni momento utilizzazione massima, creando attese considerevoli ed inducendo gli utenti ad abbandonare da subito l'uso del social network.

Definiamo adesso i parametri usati come input al sistema, partendo dai tassi di arrivo λ, suddivisi in 1 e 2 in base al numero di task in questione.

$$\lambda_1 = 1.5j/s \qquad \lambda_2 = 1.8j/s$$

Nell'intero sistema i task in entrata possono essere considerati come un flusso unico, ciascuno pesato singolarmente attraverso la propria percentuale rispetto al totale, da cui si ricava: $\lambda_{tot} = \lambda_1 + \lambda_2$

La percentuale di arrivo dei task nel sistema, differente in base al tipo, assume i seguenti valori, calcolati con le rispettive formule:

$$p_1 = \frac{\lambda_1}{\lambda_{tot}} = \frac{1.5s^{-1}}{3.3s^{-1}} = 0.454545$$

$$p_2 = \frac{\lambda_2}{\lambda_{tot}} = \frac{1.8s^{-1}}{3.3s^{-1}} = 0.545454$$

Bisogna tenere in considerazione che una parte del λtot esce dal sistema una volta servito dalla Server Farm, mentre quella restante è l'unico flusso che attraversa le istanze di AWS EC2, di conseguenza le percentuali di task di tipo 1 e 2 variano in base al nodo preso in esame, oltre che in funzione dell'algoritmo scelto. In generale, il flusso in entrata in AWS Cloud dipende dalla *probabilità di blocco della Server Farm*, ovvero la percentuale di task che consideriamo persi nel primo nodo perché inviati al secondo. Sia $P_q$ la probabilità di blocco o perdita, si può dunque affermare:

$$\lambda_{serverfarm} = (1 - P_q) \cdot \lambda_{tot}$$

$$\lambda_{awscloud} = P_q \cdot \lambda_{tot}$$

$$\lambda_{tot} = \lambda_{serverfarm} + \lambda_{awscloud}$$

Per quanto riguarda gli arrivi, i job sono di due tipologie distinte e vengono serviti a rate differenti sia dalla server farm che dalle istanze di AWS Cloud EC2. Queste richieste in arrivo approdano al controllore della server farm, che si occupa poi di smistarle verso il nodo più opportuno: esse sono di distribuzione esponenziale e, in questo caso particolare, gli interarrivi sono poissoniani. Ciò che arriva alle istanze di AWS EC2 sono tutte quelle richieste che non sono state servite dalla server farm, sono dunque dipendenti dallo stato di quest'ultima in questo modo:

$Arrivi_{awscloud} = arrivi_{serverfarm} \cdot P_q$ con Pq definita come sopra, ossia probabilità di blocco della serverfarm.

Per la proprietà di assenza di memoria, propria della distribuzione esponenziale, gli arrivi dei vari task al sistema sono da considerarsi tra loro indipendenti (ad esclusione di quello che arriva ad AWS Cloud che, come appena scritto, è dipendente da ciò che i vari server della pool non riescono a smaltire).

Per i serventi facenti parte della server farm, abbiamo assunto che i tempi di servizio vengano generati a partire da una distribuzione iperesponenziale, prendendo come riferimento una delle tracce di progetto degli anni passati in cui veniva assunto per ipotesi. Dalla teoria si sa che, per una distribuzione iperesponenziale di parametro μ con due stati si ha: il primo stato con distribuzione esponenziale di parametro $\mu 1_{stato} = 2 \cdot p1_{fase} \cdot \mu$ con probabilità $p1_{fase}$; il secondo stato con distribuzione esponenziale di parametro $\mu 2_{stato} = 2 \cdot p2_{fase} \cdot \mu$ con probabilità $p2_{fase} = (1 - p1_{fase})$. Da ciò si evince che la media delle frequenze di servizio di un singolo server è $\mu = \frac{\mu 1_{stato} + \mu 2_{stato}}{2}$.

Nel caso preso in esame, abbiamo assunto che la generica μ di ogni server della server farm fossero:

$\mu 1_{serverfarm} = 0.2s^{-1}$ per le letture (task di tipo 1)

$\mu 2_{serverfarm} = 0.15s^{-1}$ per le scritture (task di tipo 2)

entrambi con probabilità di accesso allo stato $p1_{fase} = 0.2$ e $p2_{fase} = 0.8.$

La Server Farm, considerata singolarmente, non è stabile e presenta perdita. Per verificarlo abbiamo calcolato il valore

dell'utilizzazione, usando il valore medio della frequenza di servizio della Server Farm stessa, che abbiamo dovuto pesare in base alla probabilità dei diversi job:

$$\mu_{serverfarm} = E[\mu_{serverfarm}] = p_1 \cdot \mu 1_{serverfarm} + p_2 \cdot \mu 2_{serverfarm}$$

Essendo un multiserver, nel nostro caso l'utilizzazione rappresenta la percentuale di server occupati all'interno della farm:

$$\rho = \frac{\lambda_{tot}}{N_{server} \cdot \mu_{serverfarm}} = \frac{\lambda_1 + \lambda_2}{10 \cdot E[\mu_{serverfarm}]} = \frac{3.3s^{-1}}{10 \cdot (0.2 \cdot 0.454545 + 0.15 \cdot 0.545454)} = 1.91052 \geq 1$$

Per calcolare la percentuale di perdita della Server Farm è stata utilizzata una catena di Markov a tempo continuo e stati discreti. Si hanno N stati dove il singolo stato (i,j) della catena è dato dal numero di task 1 e 2 presenti in un dato istante. La connessione tra due stati è chiamata frequenza di transizione di stato: spostandosi da $S_k$ a $S_{k+1}$ è data dal flusso in entrata $\lambda$, mentre spostandosi da $S_{k+1}$ a $S_k$ è data dalla quantità del rate di servizio $\mu$. Tramite la distribuzione stazionaria di probabilità sono state calcolate le varie statistiche del sistema, risolvendo un sistema di equazioni ottenuto grazie a dei tagli sull'insieme degli stati, dove con taglio si intende una partizione dell'insieme di partenza. In un sistema a regime la somma dei flussi di probabilità attraverso qualunque taglio è nulla. Per ogni stato si è impostata la seguente equazione:

Flusso di probabilità entrante = flusso di probabilità uscente

Alla fine della creazione delle equazioni, è necessario aggiungere al sistema ottenuto l'equazione di normalizzazione, dovuta dalla definizione assiomatica della probabilità:

$$\sum_{(i,j) \in S} \pi_{i,j} = 1$$

Dove con S si indica l'insieme di tutti gli stati possibili che il sistema può assumere.

Ovviamente, in base all'algoritmo, è diverso il numero di stati presenti nel processo di Markov.

Una volta risolte le equazioni associate alla catena di Markov in Matlab per trovare la distribuzione di probabilità del processo, sono state calcolate le varie statistiche associate alla Server Farm, che riportiamo di seguito.

La probabilità di blocco Pq è data dalla somma del prodotto tra le probabilità di tutti gli stati in cui si ha perdita (qui viene indicato con P tale insieme), ovvero quelli in cui un nuovo arrivo comporta l'inoltro di tale task ad i servizi AWS.

$$P_q = \sum_{(i,j)\in P} \pi_{i,j}$$

Con il primo tipo di algoritmo abbiamo perdita solo negli ultimi stati della catena, mentre nel secondo è presente anche negli stati intermedi, quelli in cui (i+j) è maggiore del valore limite 8.

Dopodiché siamo passati alla probabilità di blocco per task: si considerano esclusivamente gli stati in cui si esce dalla Server Farm, perché un determinato tipo di task non può essere servito. Anche qui, come sopra, si sono indicati con P1 e P2 gli insiemi degli stati per i vari task che sono causa di perdita, rispettivamente per task 1 e 2.

$$P_{q1} = \sum_{(i,j)\in P_1} \pi_{i,j} \qquad P_{q2} = \sum_{(i,j)\in P_2} \pi_{i,j}$$

Il throughput è dato dal flusso di task che i server appartenenti alla Server Farm riescono a servire.

$\lambda_{serverfarm} = (1 - P_q) \cdot (\lambda_1 + \lambda_2)$ in totale.

$\lambda 1_{serverfarm} = (1 - P_{q1}) \cdot \lambda_1$ per quanto riguarda le letture.

$\lambda 2_{serverfarm} = (1 - P_{q2}) \cdot \lambda_2$ per quanto riguarda le scritture

I pesi dei job effettivamente serviti dalla server farm sono:

$$p1_{serverfarm} = \frac{\lambda 1_{serverfarm}}{\lambda_{serverfarm}} = \frac{(1 - P_{q1}) \cdot \lambda_1}{(1 - P_q) \cdot (\lambda_1 + \lambda_2)}$$

$$p2_{serverfarm} = \frac{\lambda 2_{serverfarm}}{\lambda_{serverfarm}} = \frac{(1 - P_{q2}) \cdot \lambda_2}{(1 - P_q) \cdot (\lambda_1 + \lambda_2)}$$

Il tempo medio di risposta della Server Farm: è dato dalla somma pesata dei tempi medi di servizio dei due tipi di task; i pesi p1 e p2 sono ricalcolati in base all'algoritmo che si usa per assegnare i task ai vari server della farm. Si parte, rispettivamente, dal totale per arrivare ai vari task:

$$E[T]_{serverfarm} = p1_{serverfarm} \cdot E[S_1]_{serverfarm} + p2_{serverfarm} \cdot E[S_2]_{serverfarm}$$

$E[T_1]_{serverfarm} = E[S_1]_{serverfarm}$ per le letture

$E[T_2]_{serverfarm} = E[S_2]_{serverfarm}$ per le scritture

dove $E[S_1]_{serverfarm} = \frac{1}{\mu 1_{serverfarm}} = 5s$ e

$$E[S_2]_{serverfarm} = \frac{1}{\mu 2_{serverfarm}} = 6.66666s$$

Il numero medio di task nella Server Farm viene calcolato sfruttando il teorema di Little, nel seguente modo.

$$E[N]_{serverfarm} = \lambda_{serverfarm} \cdot E[T]_{serverfarm}$$

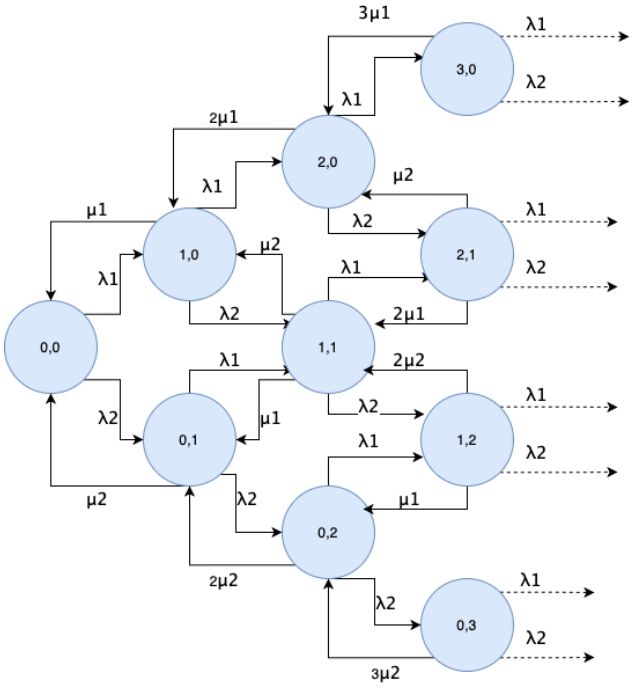
$$E[N_1]_{serverfarm} = \lambda 1_{serverfarm} \cdot E[T_1]_{serverfarm}$$

$$E[N_2]_{serverfarm} = \lambda 2_{serverfarm} \cdot E[T_2]_{serverfarm}$$

Riportiamo adesso delle raffigurazioni d'esempio della catena di Markov che descrive il nostro modello per entrambi gli

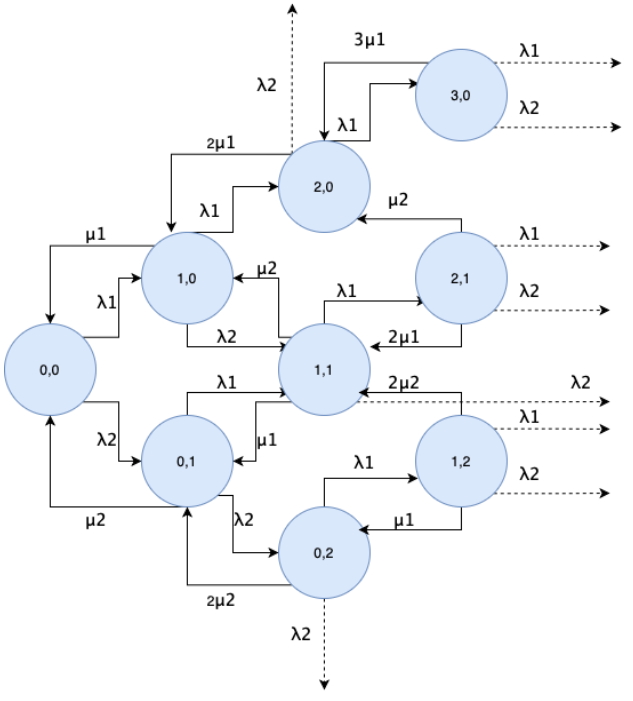
algoritmi: tutti i casi di perdita indicano l'inoltro di quel task alle istanze EC2 da parte del controllore. In questo esempio, si assume il caso particolare in cui N sia uguale a 3 e la soglia limit sia fissata a 2, come indicano le rispettive didascalie.

## Algoritmo 1:




Caso particolare di Catena di Markov con N=3 senza soglia LIMIT
Le linee tratteggiate indicano i casi in cui si ha perdita

## Algoritmo 2:




Caso particolare di Catena di Markov con N=3 con soglia LIMIT=2 per l'algoritmo 2
Le linee tratteggiate indicano i casi in cui si ha perdita

Per quanto concerne la struttura di AWS Cloud e, quindi, delle varie istanze di macchine virtuali EC2 avviate on demand, abbiamo assunto che ciascuna di esse segua una distribuzione esponenziale con parametri:

$\mu 1_{awscloud} = 0.11 s^{-1}$ per quanto riguarda le letture (task 1)

$\mu 2_{awscloud} = 0.08 s^{-1}$ per quanto riguarda le scritture (task 2)

Le statistiche per questa parte del sistema sono state calcolate considerando come flusso in ingresso tutti quei job non presi in carico dalla server farm.

Per quanto riguarda il throughput di AWS Cloud EC2, non essendoci perdita, viene dato dal flusso entrante nel nodo stesso: il throughput totale è dato da $\lambda_{awscloud} = P_q \cdot (\lambda_1 + \lambda_2)$. Per quanto riguarda le letture abbiamo $\lambda_{awscloud1} = P_{q1} \cdot \lambda_1$, mentre per le scritture $\lambda_{awscloud2} = P_{q2} \cdot \lambda_2$. Abbiamo calcolato le percentuali di task in ingresso nel nodo come:

$$p1_{awscloud} = \frac{\lambda_{awscloud1}}{\lambda_{awscloud}} = \frac{P_{q1} \cdot \lambda_1}{P_q \cdot (\lambda_1 + \lambda_2)}$$

$$p2_{awscloud} = \frac{\lambda_{awscloud2}}{\lambda_{awscloud}} = \frac{P_{q2} \cdot \lambda_2}{P_q \cdot (\lambda_1 + \lambda_2)}$$

Per quanto riguarda i tempi di risposta, abbiamo considerato una media pesata dei tempi medi di servizio dei due tipi di serventi. Il risultato è il seguente, rispettivamente per E[T] totale, per letture e per scritture (task 1 e task 2):

$$E[T]_{awscloud} = p1_{awscloud} \cdot E[S_1]_{awscloud} + p2_{awscloud} \cdot E[S_2]_{awscloud}$$

$$E[T1_{awscloud}] = E[S_1]_{awscloud}$$

$$E[T2_{awscloud}] = E[S_2]_{awscloud}$$

in cui $E[S_1]_{awscloud} = \frac{1}{\mu 1_{awscloud}} = 9.09090s$ ed

$$E[S_2]_{awscloud} = \frac{1}{\mu 2_{awscloud}} = 12.5s$$

Per quanto riguarda il numero medio di task che si trovano nell'ambiente AWS Cloud, abbiamo utilizzato il Teorema di Little:

$$E[N]_{awscloud} = \lambda_{awscloud} \cdot E[T]_{awscloud}$$

$$E[N_1]_{awscloud} = \lambda 1_{awscloud} \cdot E[T_1]_{awscloud}$$

$$E[N_2]_{awscloud} = \lambda 2_{awscloud} \cdot E[T_2]_{awscloud}$$

Considerando, infine, il sistema nella sua interezza, abbiamo ricavato i valori sommando quelli dei singoli componenti, pesati per la probabilità di blocco ed, eventualmente, per tipo di task in arrivo. Per quanto riguarda il throughput medio del sistema:

$\lambda_{sist} = \lambda_{serverfarm} + \lambda_{awscloud}$ per il sistema nella sua totalità

$\lambda 1_{sist} = \lambda 1_{serverfarm} + \lambda 1_{awscloud}$ per i task di tipo 1.

$\lambda 2_{sist} = \lambda 2_{serverfarm} + \lambda 2_{awscloud}$ per i task di tipo 2.

Vediamo adesso il numero medio di task nel sistema:

$E[N]_{sist} = E[N]_{serverfarm} + E[N]_{awscloud}$ per il sistema totale

$E[N_1]_{sist} = E[N_1]_{serverfarm} + E[N_1]_{awscloud}$ per il task 1

$E[N_2]_{sist} = E[N_2]_{serverfarm} + E[N_2]_{awscloud}$ per il task 2.

Infine, il tempo di risposta medio del sistema, dato dalle rispettive somme pesate. Si riportano in ordine: il totale, per task 1 e per task 2.

$$E[T]_{sist} = (1 - P_q) \cdot E[T]_{serverfarm} + P_q \cdot E[T]_{awscloud}$$

$$E[T_1]_{sist} = (1 - P_{q1}) \cdot E[T_1]_{serverfarm} + P_{q1} \cdot E[T_1]_{awscloud}$$

$$E[T_2]_{sist} = (1 - P_{q2}) \cdot E[T_2]_{serverfarm} + P_{q2} \cdot E[T_2]_{awscloud}$$

# Modello computazionale

Per generare gli eventi randomicamente ci si è basati sul generatore di Lehmer. Un buon generatore deve garantire randomicità, portabilità, controllabilità ed efficienza per poter produrre risultati che siano statisticamente indistinguibili. Nel caso in esame, l'algoritmo permette di creare degli streams di numeri ricorsivamente utilizzando tre valori: il moltiplicatore, il modulo e l'incremento. Queste tre componenti sono pressoché indipendenti tra loro, ma i numeri generati si ripetono con una certa periodicità. Un altro parametro per offrire ripetibilità nella simulazione è il "seed" (seme) che deve avere delle proprietà affinché i valori generati non seguano dei pattern.

Lo studio del nostro modello è partito proprio da questo aspetto, forse tra i più critici e rilevanti della simulazione, per poter generare un buon simulatore che estrapoli degli output corretti ed affidabili. Per selezionare un insieme di seed candidati ci siamo affidati al test del Chi-Quadro di Pearson (o della bontà dell'adattamento). Si tratta di un test non parametrico applicato a grandi campioni per verificare se quello preso in esame è stato estratto da una popolazione con una predeterminata distribuzione o che due o più campioni derivino dalla stessa popolazione, attraverso un test di ipotesi

con accuratezza del 95%. Nel nostro caso, infatti, vogliamo verificare che i valori prodotti da un seed siano sufficientemente randomici: partendo da una sequenza di numeri provenienti dallo stesso seed, ottenuti a partire dal generatore di Lehmer, abbiamo partizionato l'insieme in "bins" equidistanti e confrontato le distribuzioni di ciascuna partizione con le altre. Ma qual è l'output di questo test? Vengono prodotti due risultati: il primo è il risultato del test calcolato per la popolazione che si sta valutando (chi_test), il secondo è il valore critico ricavato dalla distribuzione della $\chi^2$ dipendente dalla distribuzione (x). Questi output li abbiamo usati per formulare la nostra ipotesi nulla e l'ipotesi alternativa:

$H_0$: chi_test < x $\qquad\qquad$ $H_1$: chi_test $\geq x$

Se l'ipotesi alternativa viene confermata, è necessario rigettare l'ipotesi nulla e, quindi, considerare il seed osservato come inadatto.

Il nostro simulatore genera tempi di servizio e arrivo utilizzando il seed scelto "12345". Ogni nuova replicazione della simulazione utilizza la funzione plantSeeds(r.getSeed()). Per costruzione la funzione r.getSeed() ritorna un seme diverso da quello di partenza dopo che si è utilizzata la funzione r.random() all'interno della simulazione/replicazione precedente (si precisa che la funzione random è stata usata per generare i tempi di servizio e arrivo). Per valutare la bontà di questi seed generati abbiamo scelto di raccoglierli producendo delle popolazioni, per ognuno di essi, utilizzando la funzione r.random(). Il seguente test è stato utile per verificare che le popolazioni fossero abbastanza randomiche singolarmente. Riportiamo ora i semi riferiti alle 5 replicazioni per l'algoritmo1 del transient-state.

```
seme: Seed1202331895, chi_test: 67.878600, chi_quadro:117.406883
seme: Seed1808600557, chi_test: 81.365400, chi_quadro:117.406883
seme: Seed1994218610, chi_test: 70.335200, chi_quadro:117.406883
seme: Seed2060527362, chi_test: 77.879400, chi_quadro:117.406883
seme: Seed2139392116, chi_test: 72.991000, chi_quadro:117.406883
```

In questo test si può osservare che il seed scelto inizialmente (12345) non compare. Questo perché l'analisi transiente è stata runnata con diversi valori di stop, in particolare con 8 valori incrementali da 5.00 a 10000.00 (valori contenuti in STOP_T) e, per ognuno di questi valori, sono state prodotte 5 replicazioni. Per ogni replicazione e per ogni valore di STOP_T viene, quindi, generato un nuovo seed come descritto in precedenza. Il test sopra riportato si riferisce all'ultimo valore dello STOP_T.

Dal test è possibile notare come il valore critico, la nostra ipotesi alternativa H1 , sia sempre lo stesso essendo dipendente dalla distribuzione della popolazione.
Questa verifica ci ha permesso di capire che non dovevamo scartare alcun seme, dato che non abbiamo mai dovuto rigettare l'ipotesi nulla.

La seconda verifica che abbiamo effettuato è quanto i seed fossero correlati tra loro e, di conseguenza, quanto la dipendenza tra le popolazioni possa influire sulle nostre simulazioni. Per testarne l'indipendenza abbiamo utilizzato il Test di Wilcoxon: un test di ipotesi non parametrico, per cui non è necessario conoscere il tipo di distribuzione che hanno i dati, che verifica l'indipendenza dei dataset, basandosi sempre su un test di ipotesi con accuratezza del 95%. Le assunzioni per poter applicare il test sono le seguenti: i valori confrontati sono indipendenti, che è anche l'ipotesi nulla; i valori sono confrontabili e la distribuzione, anche se non conosciuta, è continua.

Continuando ad usare gli stessi streams di numeri, generati precedentemente dal generatore di Lehmer, abbiamo confrontato i risultati del test di Wilcoxon.
Il test di ipotesi utilizzato dal Test di Wilcoxon considera:
$H_0$: le differenze calcolate tra i valori del dataset, individuo per individuo, provengono da una popolazione con media nulla;
$H_1$: la popolazione normalizzata non ha media nulla.
Nel nostro caso la popolazione utilizzata dal test viene prima

ottenuta calcolando le distanze tra i due dataset per poi verificare che la distribuzione del nuovo insieme sia normale con media nulla, in modo da considerare i valori a coppie di due-a-due contemporaneamente.

| Seeds | Seed1202331895 | Seed1808600557 | Seed1994218610 | Seed2060527362 | Seed2139392116 |
|---|---|---|---|---|---|
| 'Seed1202331895' | 0 | 0 | 0 | 0 | 0 |
| 'Seed1808600557' | 0 | 0 | 0 | 0 | 0 |
| 'Seed1994218610' | 0 | 0 | 0 | 0 | 0 |
| 'Seed2060527362' | 0 | 0 | 0 | 0 | 0 |
| 'Seed2139392116' | 0 | 0 | 0 | 0 | 0 |

A conferma di quanto detto in precedenza, anche questo test ci dimostra come nessuno dei semi da noi utilizzato sia dovuto essere scartato, non comparendo in nessuno dei campi del test il valore di h=1. Di conseguenza, l'ipotesi nulla non è mai dovuta essere rigettata.

La simulazione si basa sul modello discreto *next-event*: è presente un orologio asincrono che scorre gli eventi fino al completo esaurimento degli stessi, saltando i tempi vuoti. Per evento si intende un qualsiasi fenomeno che concorre a cambiare lo stato del sistema in questione e, nel nostro caso, gli eventi possibili sono 3: un arrivo, che viene assegnato al primo servente disponibile tra quelli della server farm o di AWS EC2, un servizio prodotto dalla server farm, che comporta la fuoriuscita del task dal sistema o un servizio prodotto da un'istanza EC2, che comporta lo stesso effetto appena descritto.

Per simulare la distribuzione degli arrivi, abbiamo considerato il flusso globale in ingresso e una distribuzione bernoulliana con probabilità di successo pari al rapporto tra i tassi di arrivi del task1 e il tasso totale degli arrivi, mentre la probabilità di insuccesso rappresenta l'evento complementare, ovvero l'arrivo di un task2.

Il Server Farm è costituito da 10 server che globalmente rappresentano un multiserver , ognuno con distribuzione iperesponenziale.
Come descritto in precedenza, un server con distribuzione iperesponenziale è composto da due fasi con distribuzione esponenziale, con una probabilità $p1$ di accedere al primo servente e $p2 = (1 - p1)$ di accedere al secondo.

L'ambiente AWS Cloud EC2 è costituito da infinite macchine virtuali, ciascuna con il comportamento di un single-server senza coda e con tempo di servizio generato a partire da una distribuzione esponenziale.

L'avanzamento temporale nel sistema è rappresentato attraverso una classe incrementale, chiamata SystemClock. che simula il progredire del tempo selezionando il prossimo evento e, di conseguenza, il successivo valore temporale.

La classe su cui si fonda il simulatore è l'oggetto di tipo EventState.

Nel simulatore sono presenti tre diverse classi che mantengono le statistiche aggregate dei nodi: ServerFarm, AwsEc2 e GlobalStatistics. L'utilizzo è lo stesso per tutte e tre le strutture che salvano, istante per istante, il prodotto tra il numero medio di task presenti in un dato nodo e l'incremento temporale tra due eventi successivi: la somma per tutti gli istanti rappresenta l'area integrale della funzione che lega il numero di task al tempo. Essendo un simulatore Next-Event, un istante è dato dalla differenza di tempo tra due eventi successivi.

Per ottenere delle misurazioni effettive di tutte le statistiche stimate analiticamente, abbiamo implementato una versione del simulatore per analizzare la fase transiente "*finite horizon*" (ad orizzonte finito o "terminating simulation") per eventi discreti, dato che le statistiche sono ancora dipendenti dall'istante in cui vengono rilevate. Sono due gli elementi per

l'analisi transiente del sistema: "*STOP_T*": un insieme di valori di STOP, ovvero il valore che il clock deve assumere per terminare una run di simulazione transiente e "transient_replications" : numero N di replicazioni di runs per singolo stop.
Si effettua un ciclo esterno sui valori di stop assegnati e uno interno sul numero di replicazioni quindi, per ognuna di queste ultime, si genera una nuova istanza dell'algoritmo scelto avendo N replicazioni dell'esperimento per ogni *STOP_T* .
Ogni nuova istanza del simulatore utilizza un seed generato dalla run precedente, in modo da evitare possibili sovrapposizioni tra i semi e considerare le varie prove ancora indipendenti. Da questa fase della simulazione si ottiene e memorizza un campione di valori per ogni statistica per poi, successivamente, generare i rispettivi intervalli di confidenza. È fondamentale notare che, in questa fase di verifica, lo stato iniziale della simulazione parte sempre dal sistema vuoto.

Per quanto riguarda l'analisi stazionaria del modello ci siamo affidati al metodo *"Batch-Means"*. In esso è prevista la suddivisione di un'intera lunga run del simulatore in un numero prestabilito di intervalli, chiamati appunto "batch", per ciascuno dei quali si ottiene un nuovo valore delle statistiche calcolate come nel caso transiente: l'inizio di un singolo batch $i$ è collegato direttamente al precedente $i$ -1, poiché utilizza lo stato finale $i$ -1 come il proprio iniziale. Nell'implementazione tutte le statistiche e le strutture di stato (*clock, tempi medi di attraversamento di awscloud, serverfarm* e sistema) vengono azzerate al termine di ogni batch, mantenendo esclusivamente il numero di task ancora in attesa di completamento nelle istanze EC2, nei server della farm e gli eventi nella struttura *system_event* in attesa di completamento al termine del batch. Il dataset ottenuto dai vari batch permette di calcolare un intervallo di confidenza per ogni statistica.
Tutte le misure dedotte analiticamente sono state osservate durante la simulazione, al fine di valutarne la correttezza. Sapendo che il calcolatore ha un comportamento finito e non è

in grado di replicare il concetto di randomicità, i risultati prodotti variano nell'intorno dei valori analitici corrispondenti con quello che è chiamato *intervallo di confidenza*.

Veniamo ora alla descrizione di come sono stati calcolati e costruiti gli intervalli di confidenza, per ottenere il valore della media. A partire da ogni statistica X con media $\mu$ e varianza $\sigma^2$ la varianza non è nota e si ha che:

$$\frac{\overline{X_n} - \mu}{\sqrt{\frac{\widehat{\sigma_n}^2}{n}}} \sim T_{Student}(n-1)$$

$\overline{X_n}$ media campionaria $\widehat{\sigma}_n^{\,2}$ varianza campionaria.

A questo punto, l'intervallo di confidenza (al 95%) viene calcolato con la seguente formula:

$$\left( \bar{x} - t_{n-1,\frac{\alpha}{2}} * \sqrt{\frac{\widehat{\sigma_n}^2}{n}},\ \ \bar{x} + t_{n-1,\frac{\alpha}{2}} * \sqrt{\frac{\widehat{\sigma_n}^2}{n}} \right)$$

Riportiamo un esempio di esecuzione di calcolo dell'intervallo di confidenza generato dal run dell'algoritmo 2 durante la fase transiente nella pagina seguente.

```
------------------------Confidence Intervals generated from: 8 ------------------------

Using a sample of elements and a 95% of confidence the values of the confidence intervals are:

Average response time of the serverfarm 5,398544 +/- 0,078056
Average response time of the serverfarm for task1 5,010001 +/- 0,083933
Average response time of the serverfarm for task2 6,594339 +/- 0,227122

Average response time of the aws ec2 11,861187 +/- 0,117234
Average response time of the aws ec2 for task1 9,071956 +/- 0,208581
Average response time of the aws ec2 for task2 12,498555 +/- 0,164428

Average response time of the system 8,814935 +/- 0,044945
Average response time of the system for task1 5,891490 +/- 0,131148
Average response time of the system for task2 11,246128 +/- 0,110647



------------------------Confidence Intervals generated from: 8 ------------------------

Using a sample of elements and a 95% of confidence the values of the confidence intervals are:

Average tasks number of the serverfarm 8,380757 +/- 0,040750
Average tasks number of the serverfarm for task1 5,868554 +/- 0,081569
Average tasks number of the serverfarm for task2 2,512204 +/- 0,059318

Average tasks number of the aws ec2 20,659692 +/- 0,269106
Average tasks number of the aws ec2 for task1 2,945497 +/- 0,188524
Average tasks number of the aws ec2 for task2 17,714194 +/- 0,269741

Average tasks number of the system 29,040449 +/- 0,293604
Average tasks number of the system for task1 8,814051 +/- 0,259307
Average tasks number of the system for task2 20,226398 +/- 0,313406



------------------------Confidence Intervals generated from: 8 ------------------------

Using a sample of elements and a 95% of confidence the values of the confidence intervals are:

Throughput  of the serverfarm 1,552523 +/- 0,014993
Throughput  of the serverfarm for task1 1,171397 +/- 0,003525
Throughput  of the serverfarm for task2 0,381126 +/- 0,012175

Throughput  of the aws ec2 1,741896 +/- 0,030500
Throughput  of the aws ec2 for task1 0,324548 +/- 0,014359
Throughput  of the aws ec2 for task2 1,417348 +/- 0,019568

Throughput  of the system 3,294418 +/- 0,017160
Throughput  of the system for task1 1,495945 +/- 0,011234
Throughput  of the system for task2 1,798474 +/- 0,011593
```

Si riportano infine i risultati analitici prodotti al termine della simulazione, partendo dalle probabilità di blocco ottenute:

Probabilità di blocco

| | Algoritmo 1 | Algoritmo 2 |
|---|---|---|
| **Pq** | 0.52754 | 0.52969 |
| **Pq1** | 0.52754 | 0.21705 |
| **Pq2** | 0.52754 | 0.79023 |
| **P1_serverfarm** | 0.45455 | 0.75671 |
| **P2_serverfarm** | 0.54545 | 0.24329 |
| **P1_awscloud** | 0.45455 | 0.18626 |
| **P2_awscloud** | 0.54545 | 0.81374 |

Si riporta adesso un esempio d'esecuzione per entrambi gli algoritmi, partendo dal primo ed arrivando al secondo, con seed selezionato "12345" e con tipo d'esperimento ad orizzonte finito in fase transiente, con il valore di "stop value" finale, ossia 10000.00. Per gli arrivi è stato utilizzato lo stream 0, per i tassi di servizio della server farm è stato utilizzato lo stream 3 e 4, per il tipo di task è stato utilizzato lo stream 5 e per i tempi di servizio di AWS Cloud lo stream 1 (si sottintende l'uso della classe rngs.java).

Le illustrazioni, che partono dall'algoritmo 1, si trovano alla pagina successiva per motivi di chiarezza.

## Algoritmo 1:

```
------------------------Results obtained from this stop value: 10000.0 ------------------------

n1_serverfarm: 7022             n2_serverfarm: 8337
n1_awsEc2: 7943         n2_awsEc2 9572

estimated lambda 3.276282
estimated lambda task 1 1.491484
estimated lambda task 2 1.784898

average tasks number in serverfarm 9.205360
average task1 number in serverfarm 3.556273
average task2 number in serverfarm 5.649088

average tasks number in aws cloud 18.856097
average task1 number in aws cloud 7.021626
average task2 number in aws cloud 11.834471

average system tasks 28.061457
average system tasks1 10.577899
average system tasks2 17.483558

Serverfarm's throughput 1.530752
Serverfarm's task1 throughput 0.699847
Serverfarm's task2 throughput 0.830906

Aws Ec2 throughput 1.745630
Aws Ec2 throughput task1 0.791638
Aws Ec2 throughput task2 0.953992

System throughput 3.2762824656781624
System task1 throughput 1.4914844127056763
System task2 throughput 1.7848977178179724
pq 0.532808
pq_1 0.530772
pq_2 0.534480

Serverfarm response time 6.013618
Serverfarm task1 response time 5.081503
Serverfarm task2 response time 6.798710

Aws Ec2 response time 10.801888
Aws Ec2 task1 response time 8.869746
Aws Ec2 task2 response time 12.405211


Average system response time 8.564847
Average task1 system response time 6.458779
Average task2 system response time 10.670105
```

| server | utilization | Task1Processed | Task2Processed |
|---|---|---|---|
| 1 | 0.920143 | 703 | 812 |
| 2 | 0.918295 | 731 | 913 |
| 3 | 0.923717 | 699 | 767 |
| 4 | 0.919771 | 746 | 850 |
| 5 | 0.916893 | 747 | 810 |
| 6 | 0.922648 | 650 | 795 |
| 7 | 0.919207 | 723 | 884 |
| 8 | 0.922339 | 672 | 856 |
| 9 | 0.921513 | 691 | 812 |
| 10 | 0.920836 | 660 | 838 |

## Algoritmo2:

```
-----------------------Results obtained from this stop value: 10000.0 -----------------------

n1_serverfarm: 11720            n2_serverfarm: 3711
n1_awsEc2: 3303         n2_awsEc2 14420

estimated lambda 3.315376
estimated lambda task 1 1.502289
estimated lambda task 2 1.813087

average tasks number in serverfarm 8.410705
average task1 number in serverfarm 5.856180
average task2 number in serverfarm 2.554525

average tasks number in aws cloud 20.995535
average task1 number in aws cloud 3.021380
average task2 number in aws cloud 17.974154

average system tasks 29.406240
average system tasks1 8.877561
average system tasks2 20.528679

Serverfarm's throughput 1.543089
Serverfarm's task1 throughput 1.171992
Serverfarm's task2 throughput 0.371097

Aws Ec2 throughput 1.772287
Aws Ec2 throughput task1 0.330298
Aws Ec2 throughput task2 1.441990

System throughput 3.3153760773568064
System task1 throughput 1.5022891599846564
System task2 throughput 1.8130869173721498
pq 0.534566
pq_1 0.219863
pq_2 0.795323
```

```
Serverfarm response time 5.450564
Serverfarm task1 response time 4.996777
Serverfarm task2 response time 6.883705

Aws Ec2 response time 11.846576
Aws Ec2 task1 response time 9.147448
Aws Ec2 task2 response time 12.464829


Average system response time 8.869654
Average task1 system response time 5.589530
Average task2 system response time 11.935297
```

| server | utilization | Task1Processed | Task2Processed |
|---|---|---|---|
| 1 | 0.838675 | 1182 | 374 |
| 2 | 0.835029 | 1195 | 384 |
| 3 | 0.841165 | 1183 | 360 |
| 4 | 0.841554 | 1115 | 378 |
| 5 | 0.838594 | 1209 | 391 |
| 6 | 0.851971 | 1072 | 360 |
| 7 | 0.848388 | 1141 | 384 |
| 8 | 0.845771 | 1186 | 340 |
| 9 | 0.838805 | 1198 | 384 |
| 10 | 0.837181 | 1239 | 356 |

# Verifica e validazione

La fase di verifica si traduce nel controllare se il simulatore sia una corretta implementazione del modello proposto, la fase di validazione si focalizza sul capire se il modello è una ragionevole rappresentazione del sistema in questione. Le fasi di verifica e validazione sono state eseguite parallelamente per motivi di efficienza.

Per la verifica del nostro simulatore, abbiamo organizzato il codice in maniera modulare in modo da facilitarne il debugging, per poi eseguire la simulazione considerando diversi parametri di input, per controllarne, infine, l'output e confrontarlo con i valori da noi calcolati manualmente durante la fase di progettazione del sistema. In questa fase si è anche verificata la presenza di eventuali errori "logici" compiuti in fase di programmazione. La modalità con la quale siamo riusciti, passo passo, a verificare che la simulazione corrispondesse realmente ai valori calcolati è stato tramite l'utilizzo del debugger di Eclipse (IDE di sviluppo utilizzato), che ci ha consentito di tenere traccia dello stato delle variabili per ogni istante di tempo. Questa operazione è stata ripetuta singolarmente per tutti e due gli algoritmi (ogni volta con valori di input differenti) e per entrambe le tecniche di simulazione utilizzate nel simulatore: transiente e batch. Per verificare la corretta implementazione del modello proposto, come già descritto nei paragrafi precedenti, abbiamo innanzitutto scongiurato, tramite i vari test d'indipendenza, che il simulatore non fosse "seed-dependent", ossia che quest'ultimo non dipendesse dalla scelta del seme per produrre output corretti. Come già affermato in precedenza, tramite questi test, è possibile garantire che le varie esecuzioni delle replicazioni della simulazione siano indipendenti tra loro.

La fase di validazione, invece, si è focalizzata sullo stabilire se il livello di accuratezza tra modello e sistema fosse rispettato, controllando se il primo rappresentasse adeguatamente il

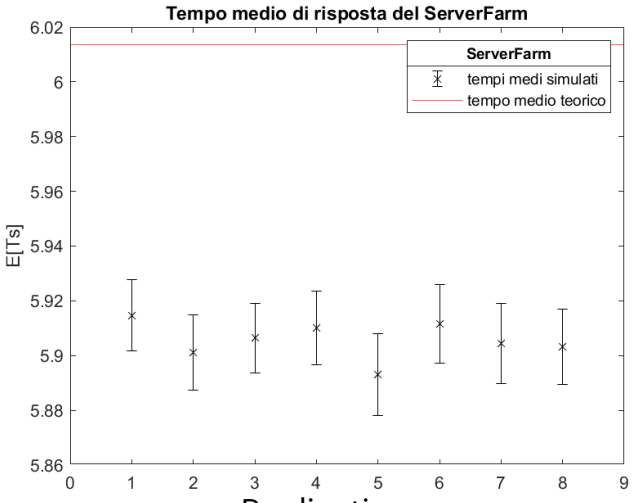
comportamento del secondo. Il confronto più significativo che si è effettuato è stato quello tra i dati ottenuti dalla simulazione e quelli che caratterizzano il sistema reale, da noi proposto, calcolati analiticamente a mano (ad esempio il valore dell'utilizzazione dei server della server farm, il valore della probabilità di blocco calcolata tramite catena di Markov...). Servendoci anche degli intervalli di confidenza, siamo arrivati alla conclusione che il nostro modello non presenta discrepanze significative tra i dati simulati e del sistema reale e risulta, di conseguenza, validato.

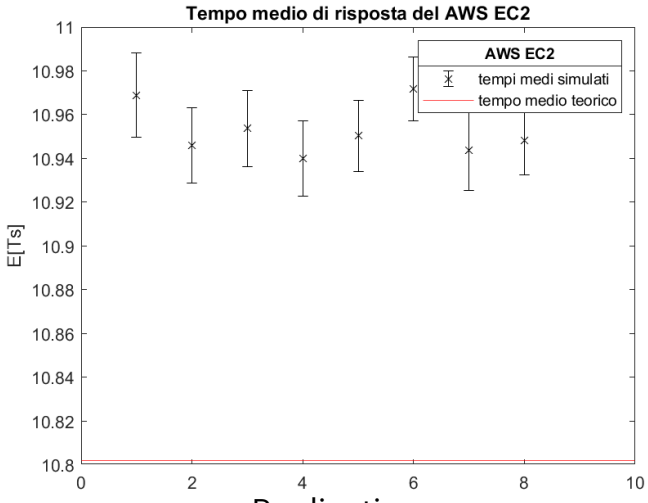
# Grafici dei risultati

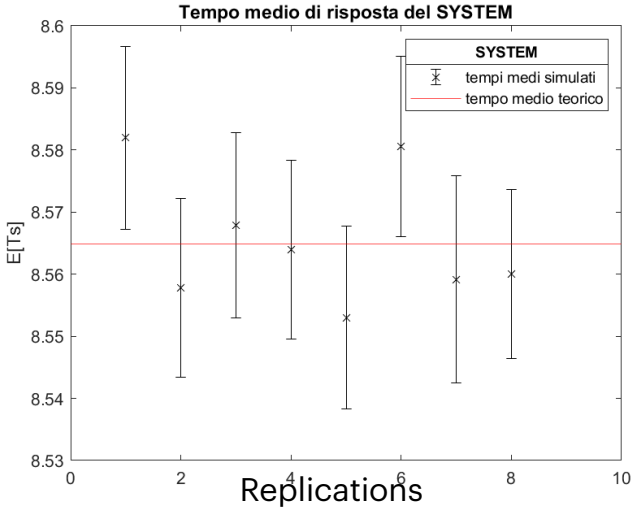
## Analisi Batch

Sono riportati di seguito tutti i grafici con i relativi intervalli di confidenza dei tempi medi di risposta, della popolazione media e del throughput trovati con il metodo *Batch-Means*.
Complessivamente il sistema risulta stazionario, come si poteva già verificare dai risultati dell'analisi ed i valori riscontrati dal modello simulativo rientrano in quelli ottenuti dal modello analitico. I grafici presentano dei problemi di visualizzazione, infatti sembra che i valori teorici si discostino da quelli ottenuti tramite la simulazione, ma vengono verificati dai valori in output dalla run.

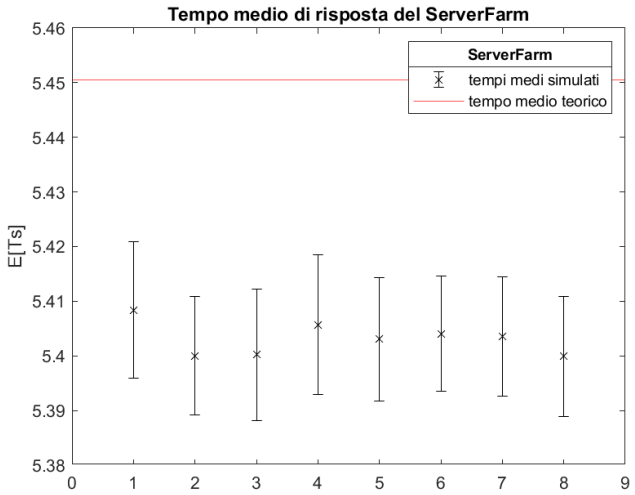
I valori ricavati per il tempo medio di risposta tramite l'algoritmo 1 sono:

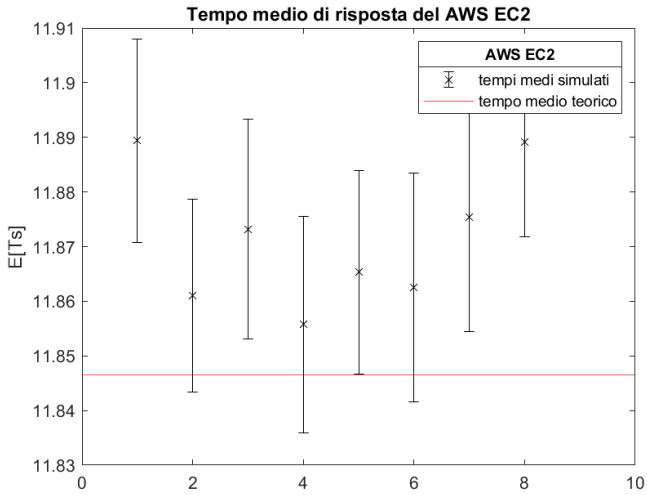


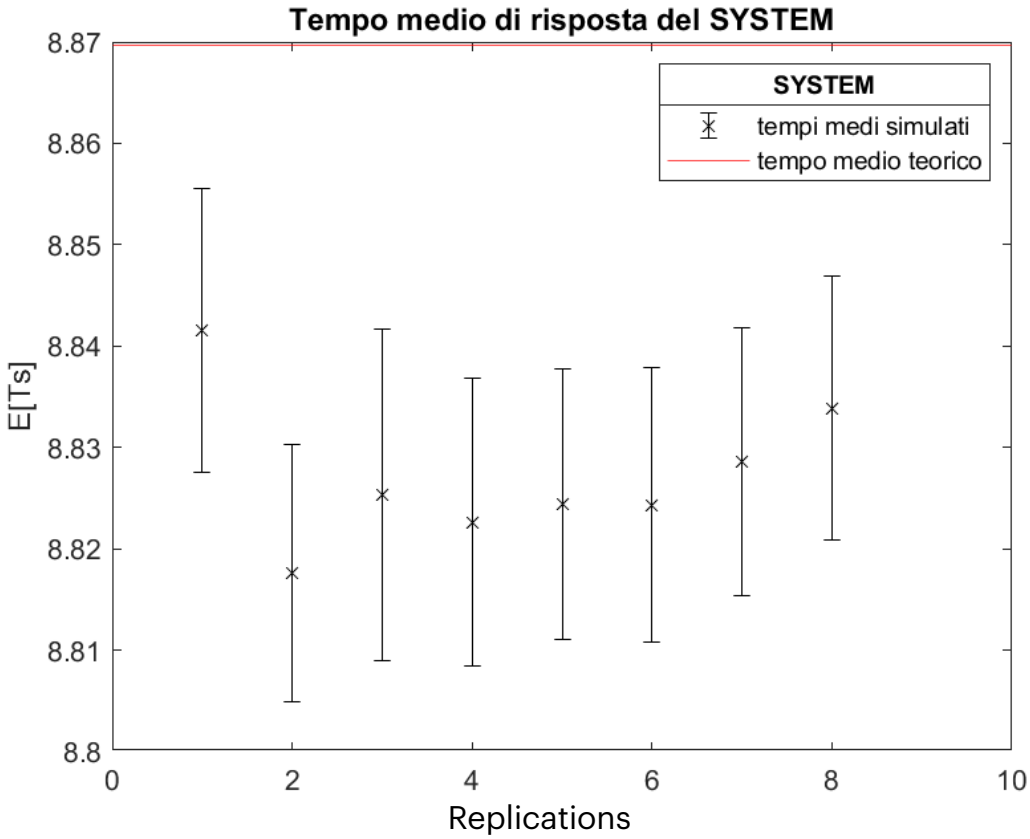



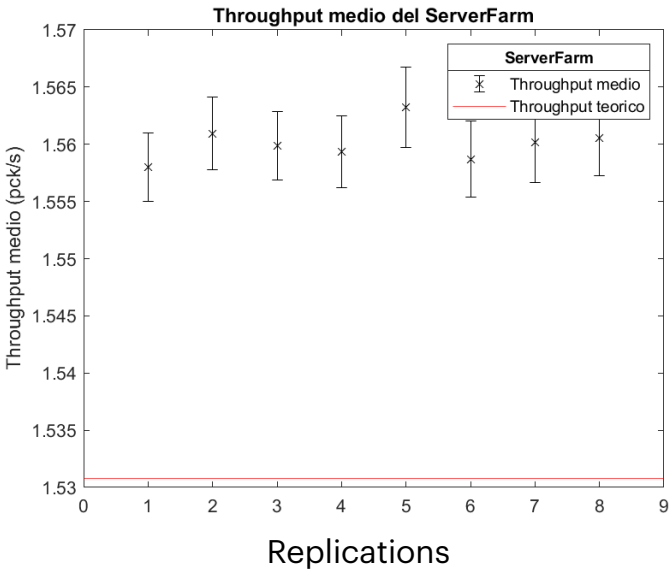
E quelli ottenuti tramite l'algoritmo 2 sono:

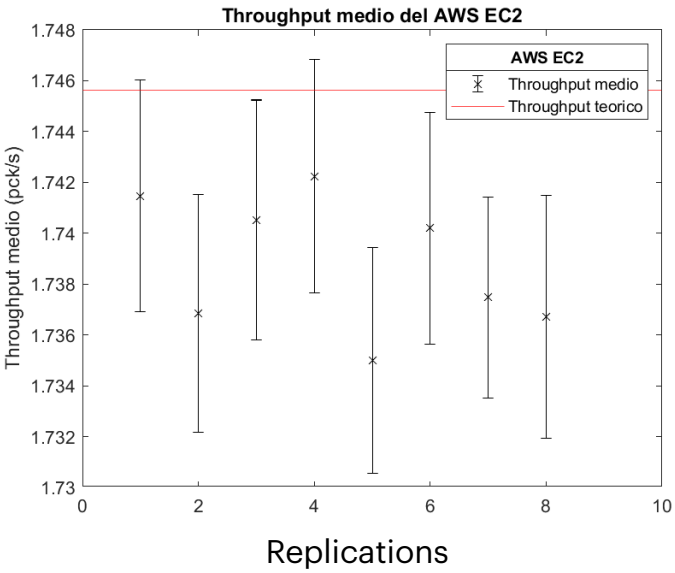


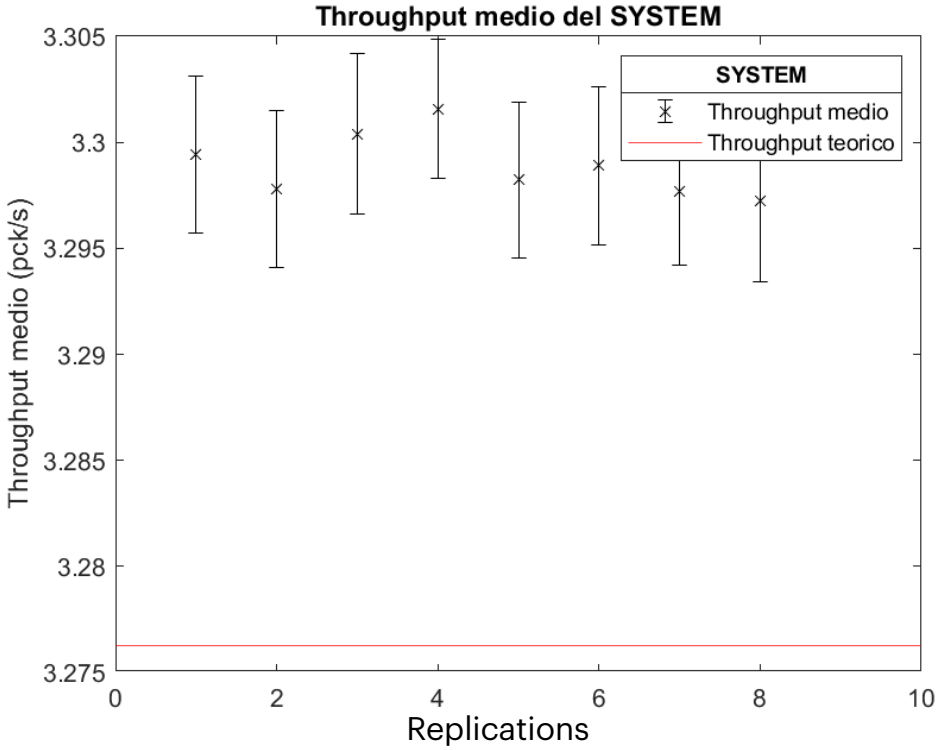



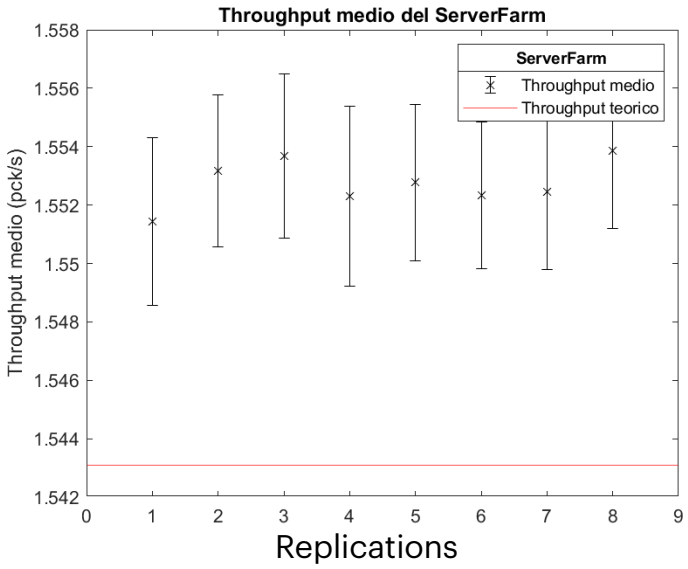
Passiamo adesso al throughput ottenuto tramite l'algoritmo 1:

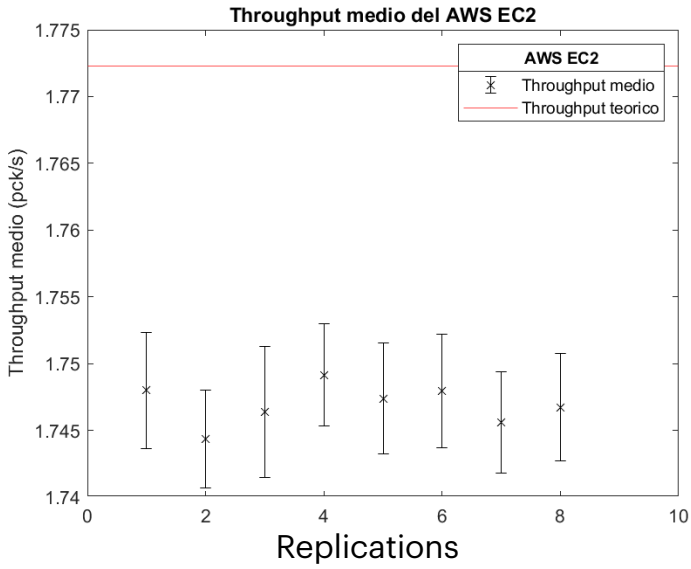


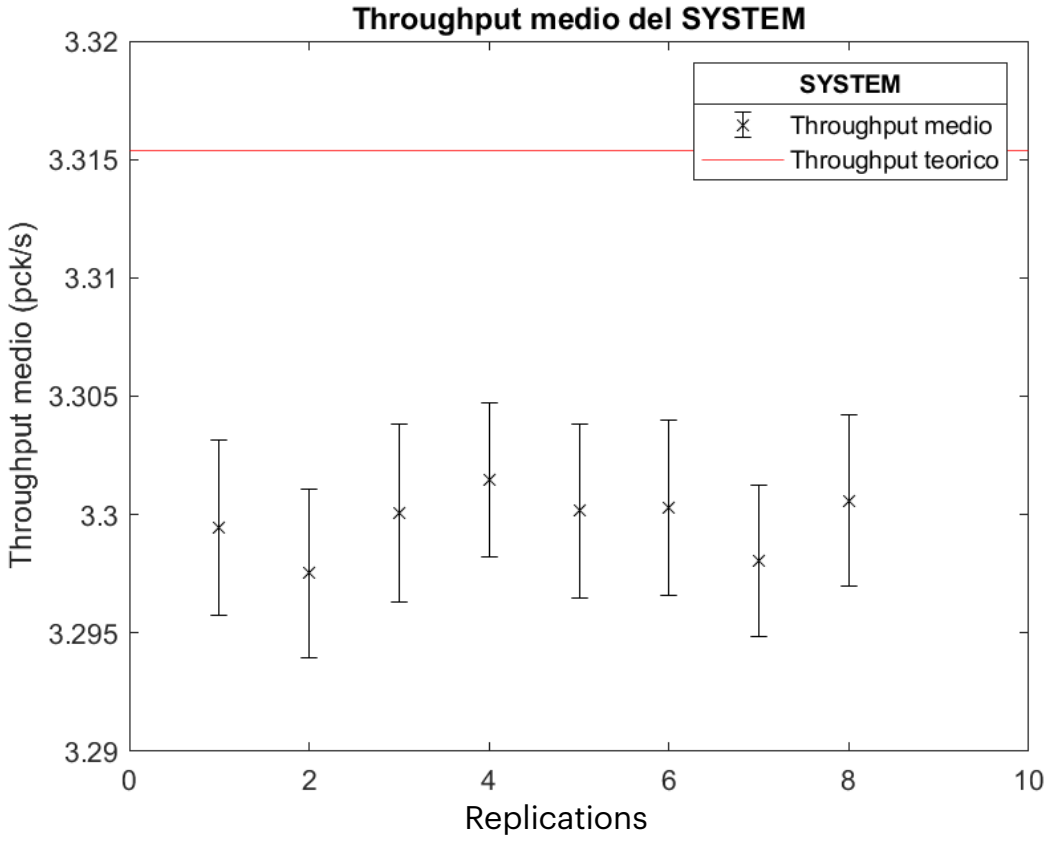



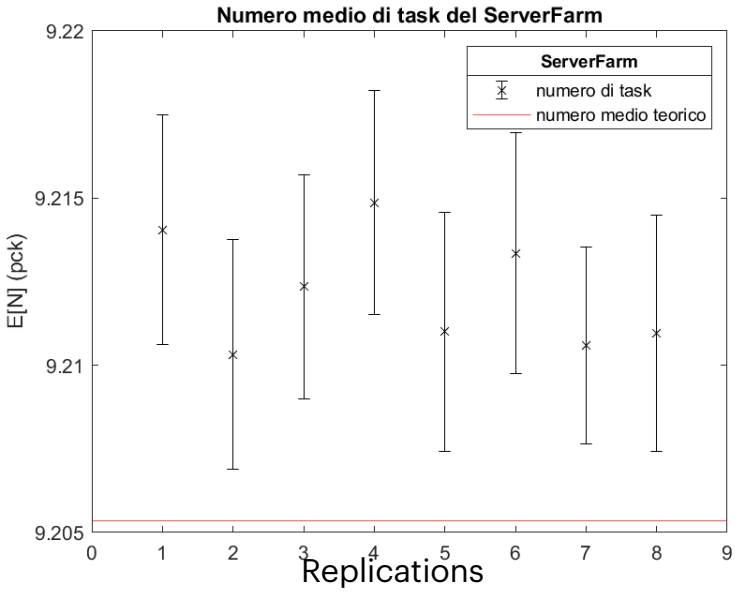
E per l'algoritmo 2:

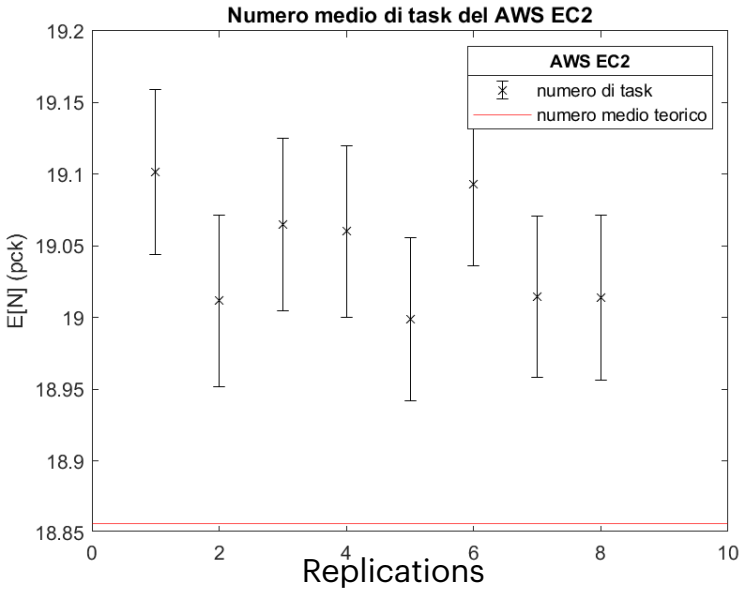


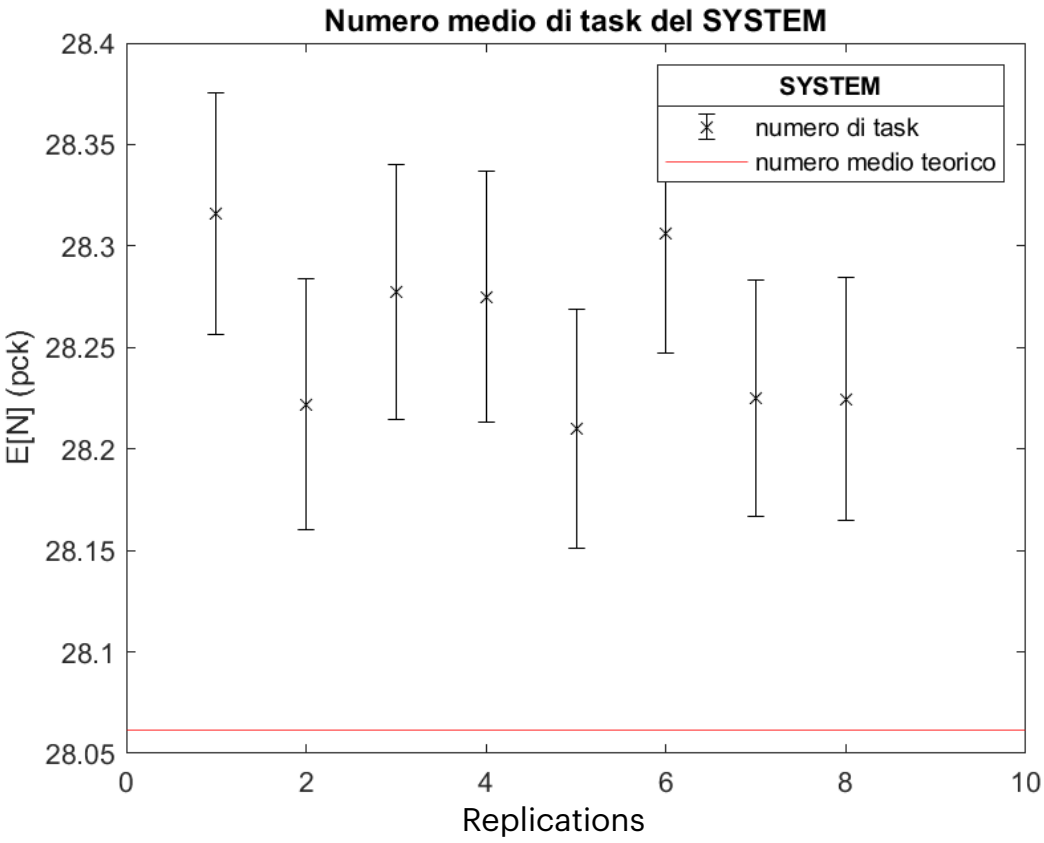



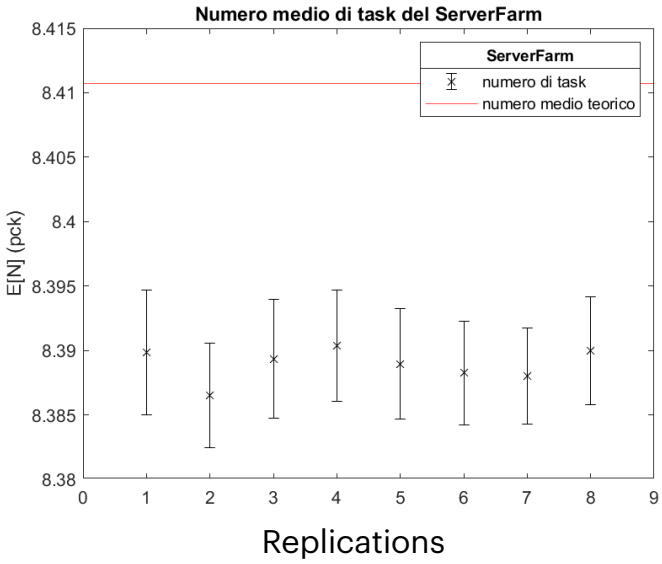
Arriviamo adesso alla popolazione media ottenuta tramite l'algoritmo 1:

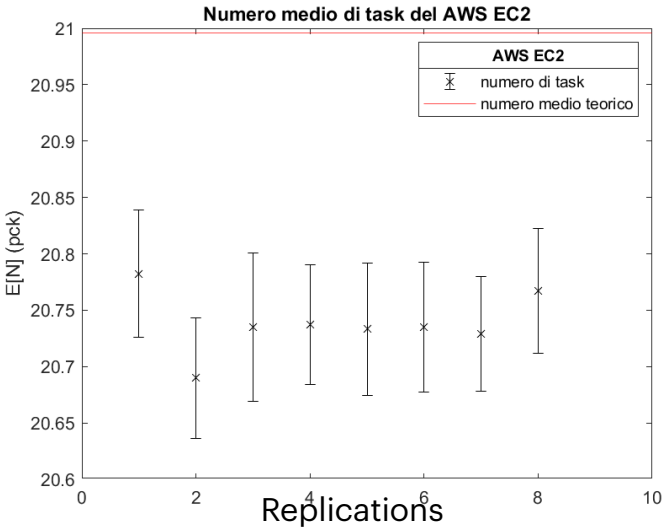


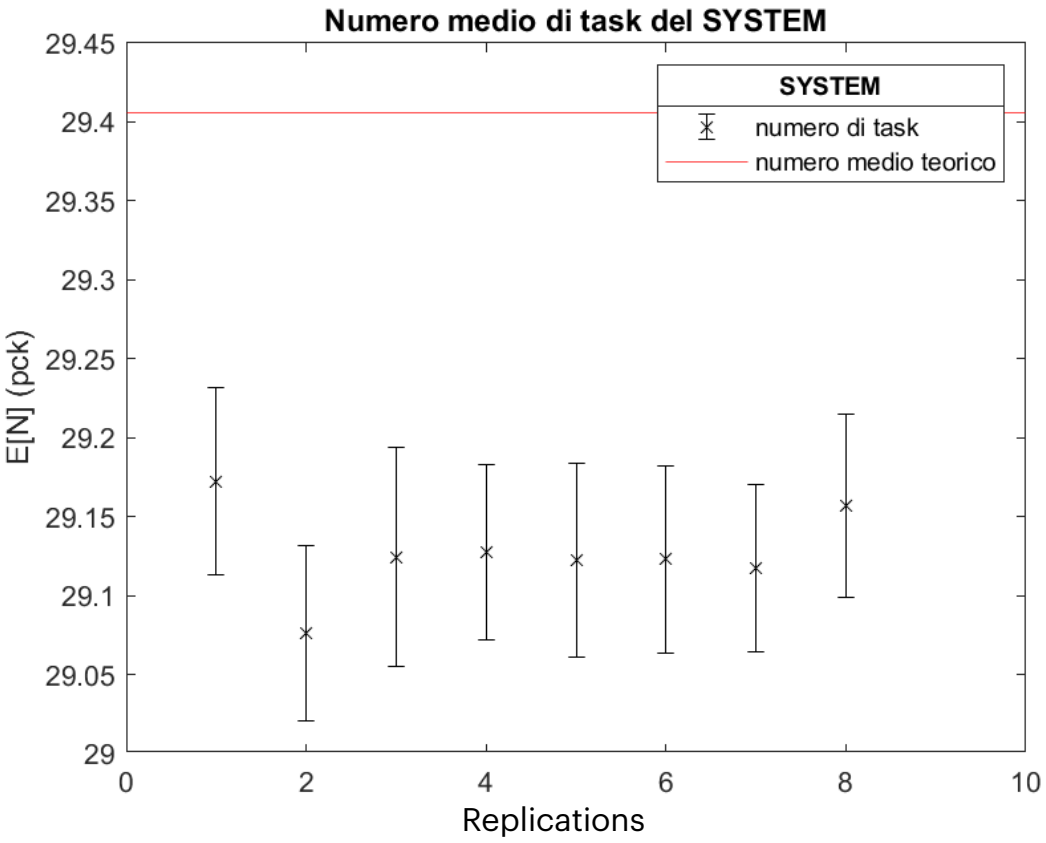



E quella ottenuta tramite l'algoritmo 2:



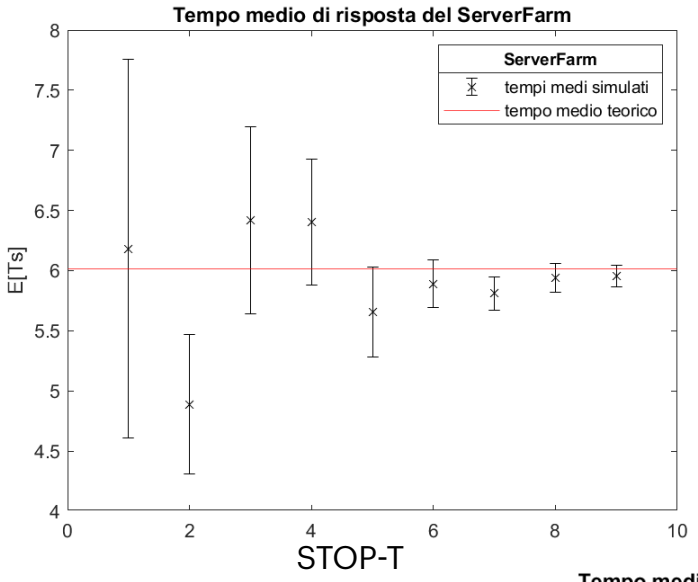



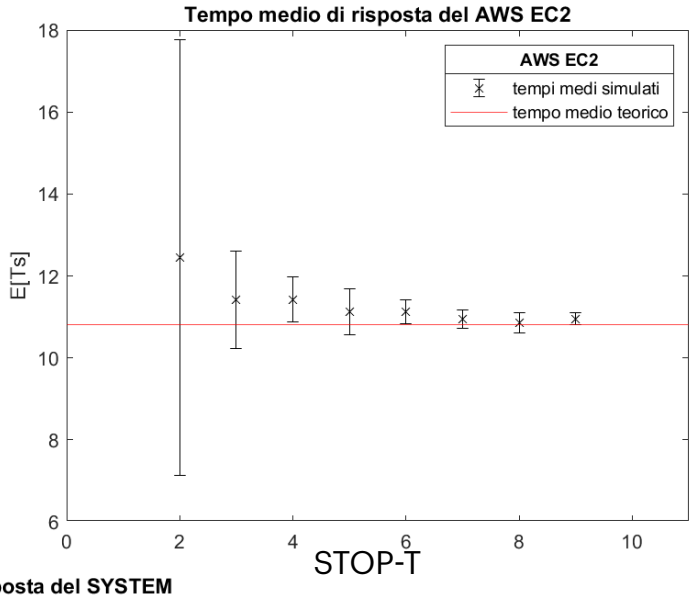



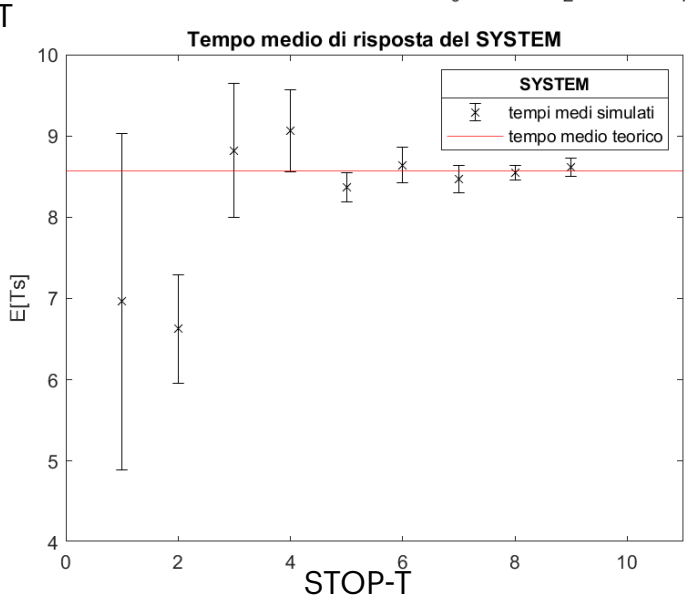
## Analisi transiente

Di seguito sono riportati i valori del tempo medio di risposta, della popolazione media e del throughput, con i relativi intervalli di confidenza, per i due algoritmi in diverse fasi d'esecuzione in cui si suppone che il sistema non abbia ancora raggiunto la stazionarietà.
A conferma dei risultati analitici, all'aumentare della durata della simulazione i risultati prodotti tendono proprio ai valori trovati.

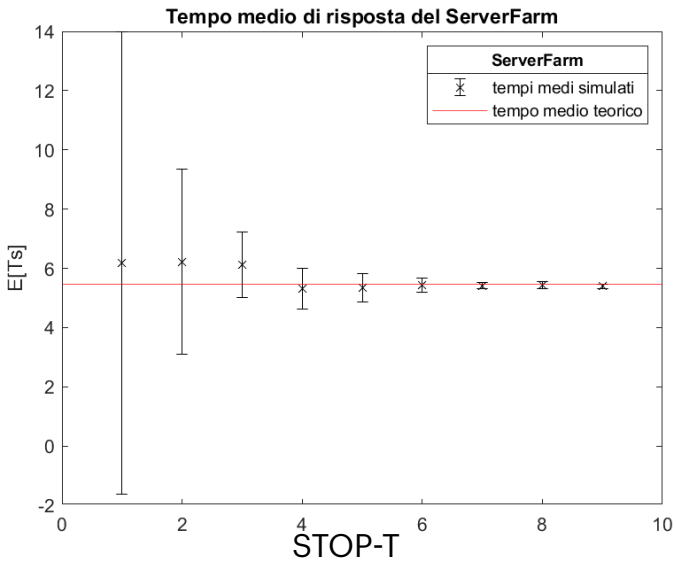
Per quanto riguarda i tempi medi di risposta ottenuti attraverso l'algoritmo 1 abbiamo:

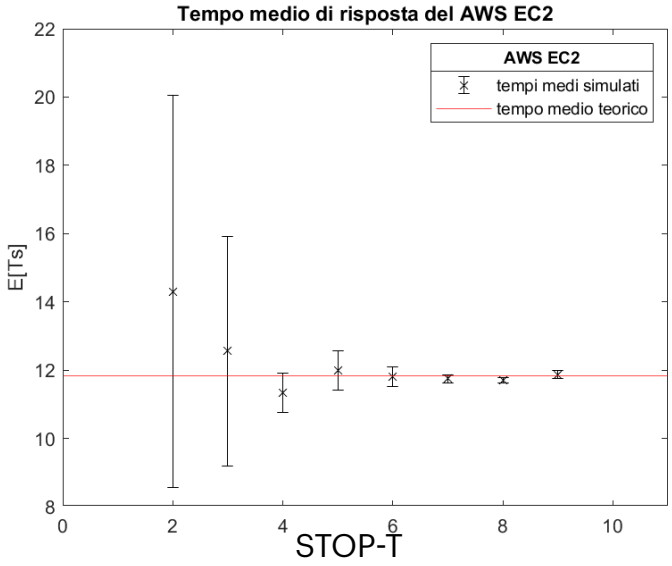


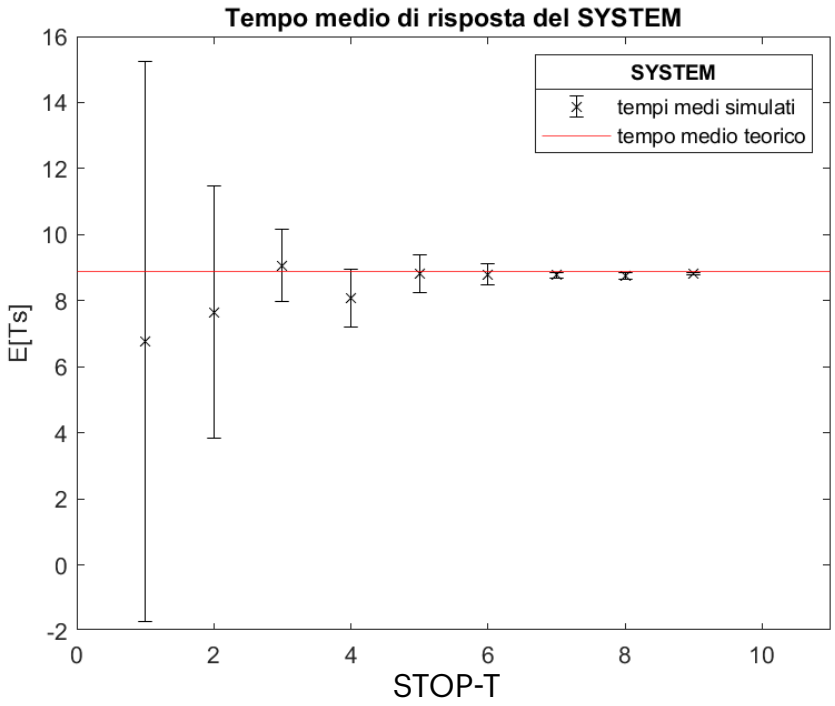



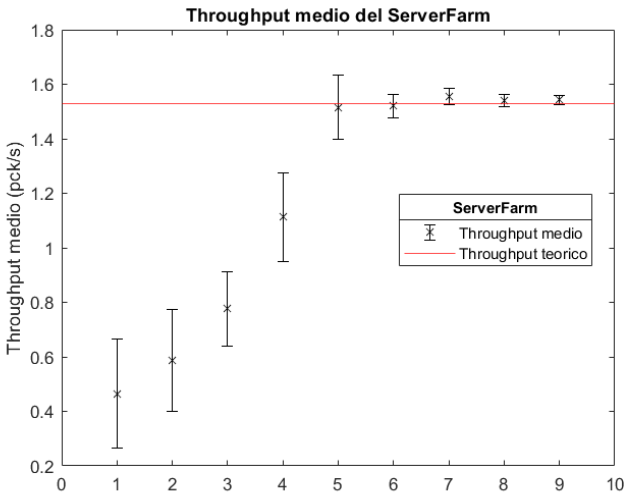
Mentre quelli dell'Algoritmo 2 sono:



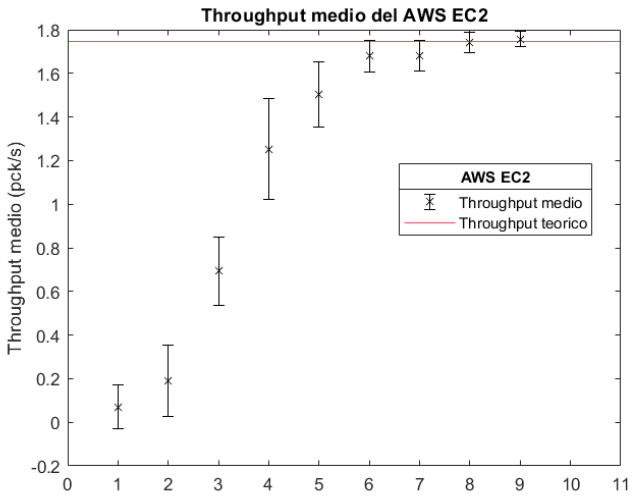



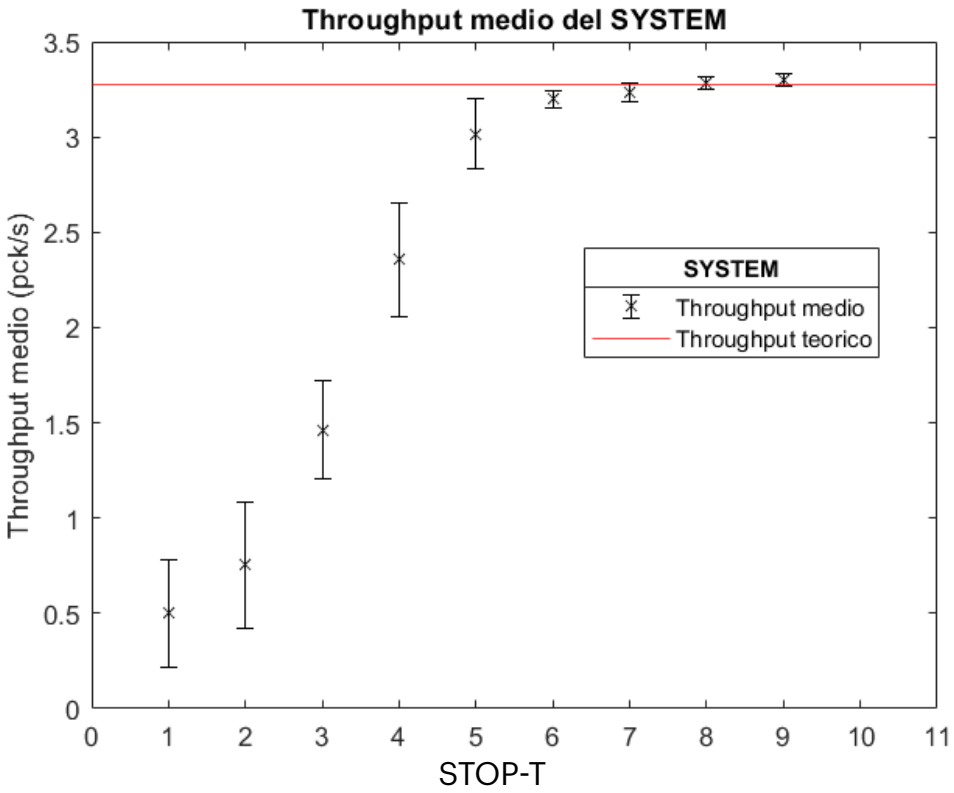
Veniamo adesso all'analisi del throughput per l'algoritmo 1:




STOP-T

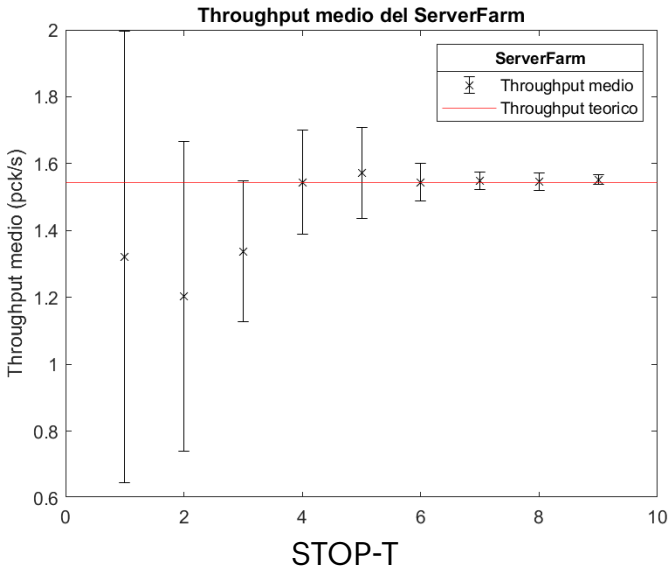


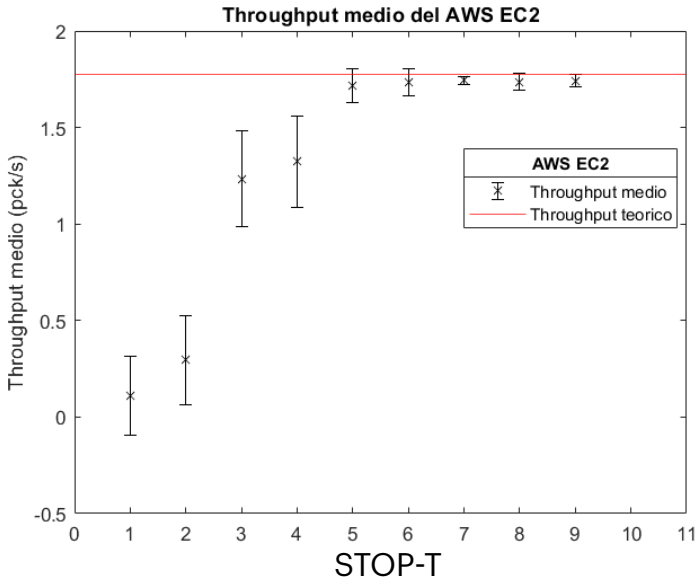

STOP-T

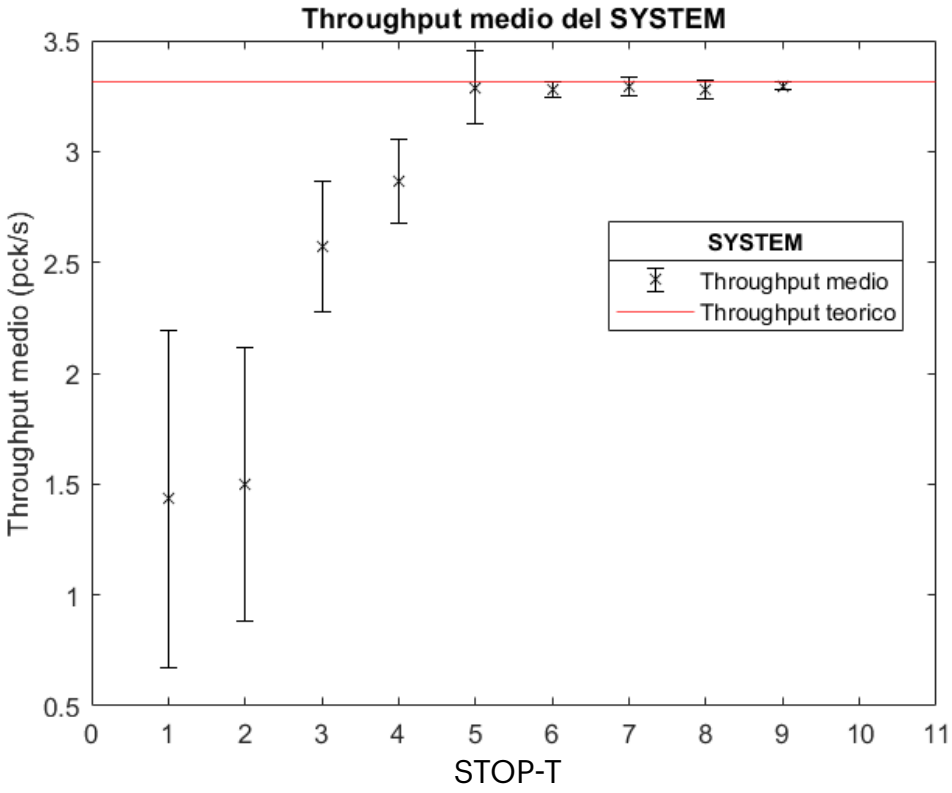



STOP-T

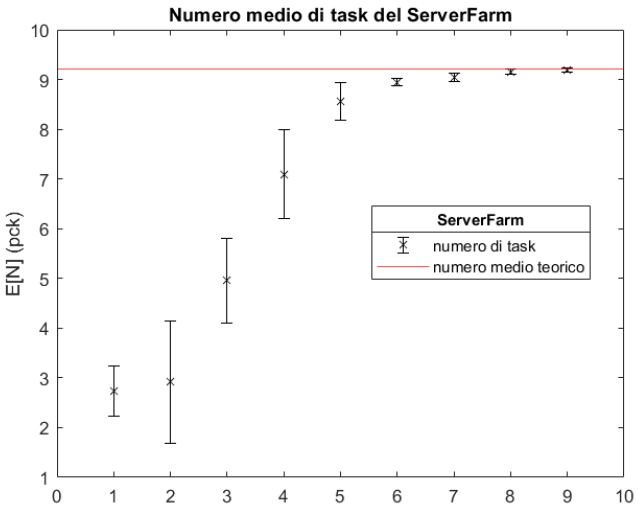
Ed invece, per l'algoritmo 2:



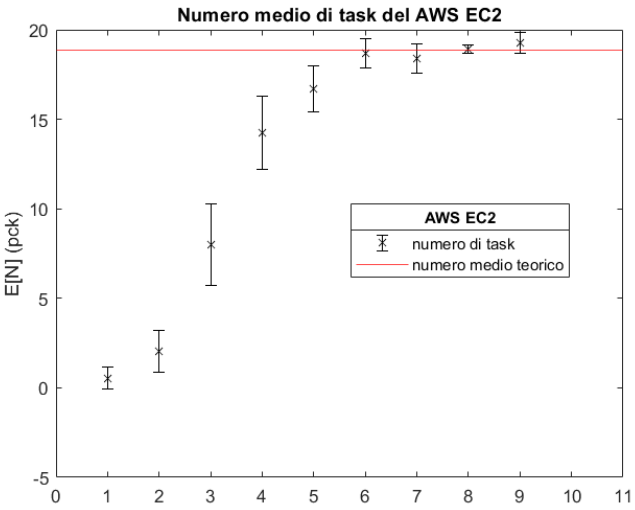



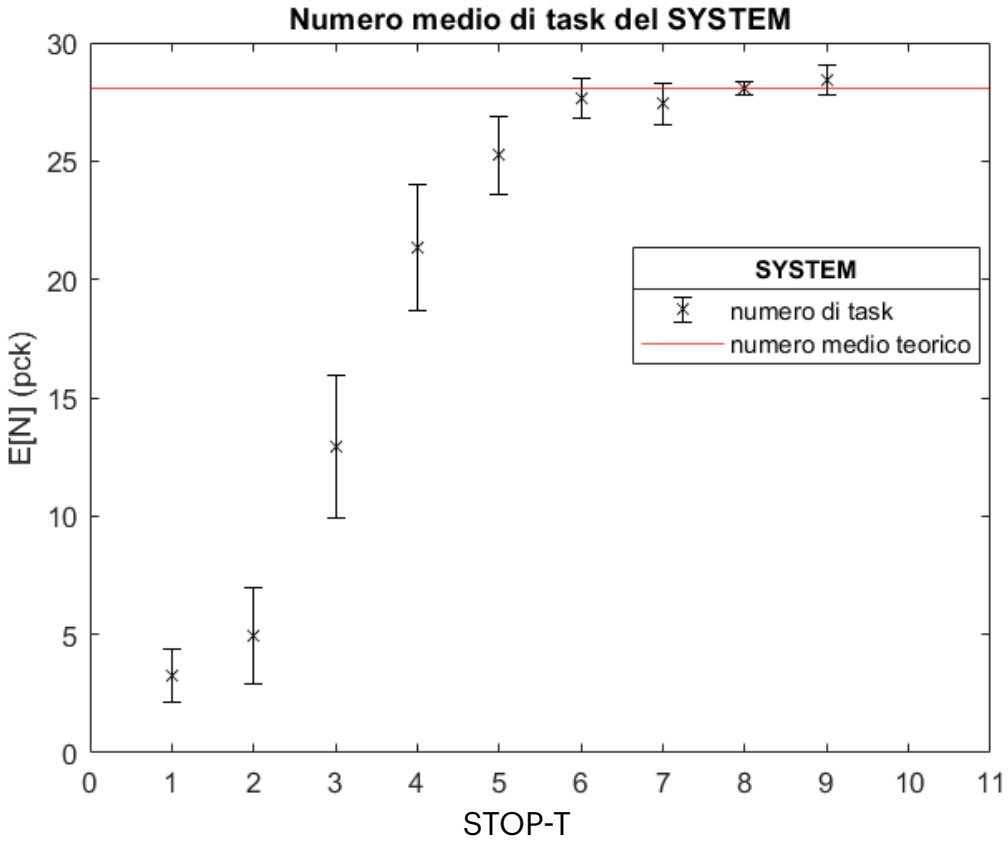
Adesso vediamo la popolazione media per l'algoritmo 1:




STOP-T

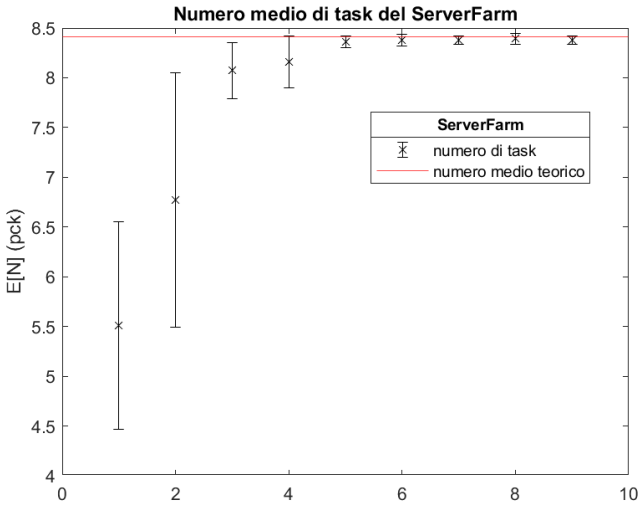



STOP-T

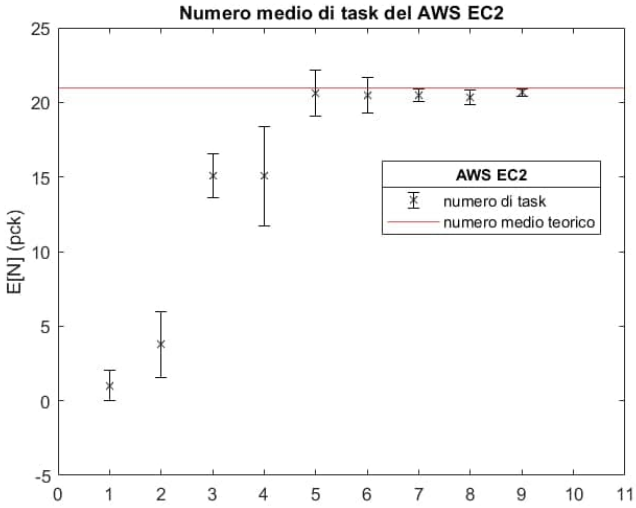



STOP-T

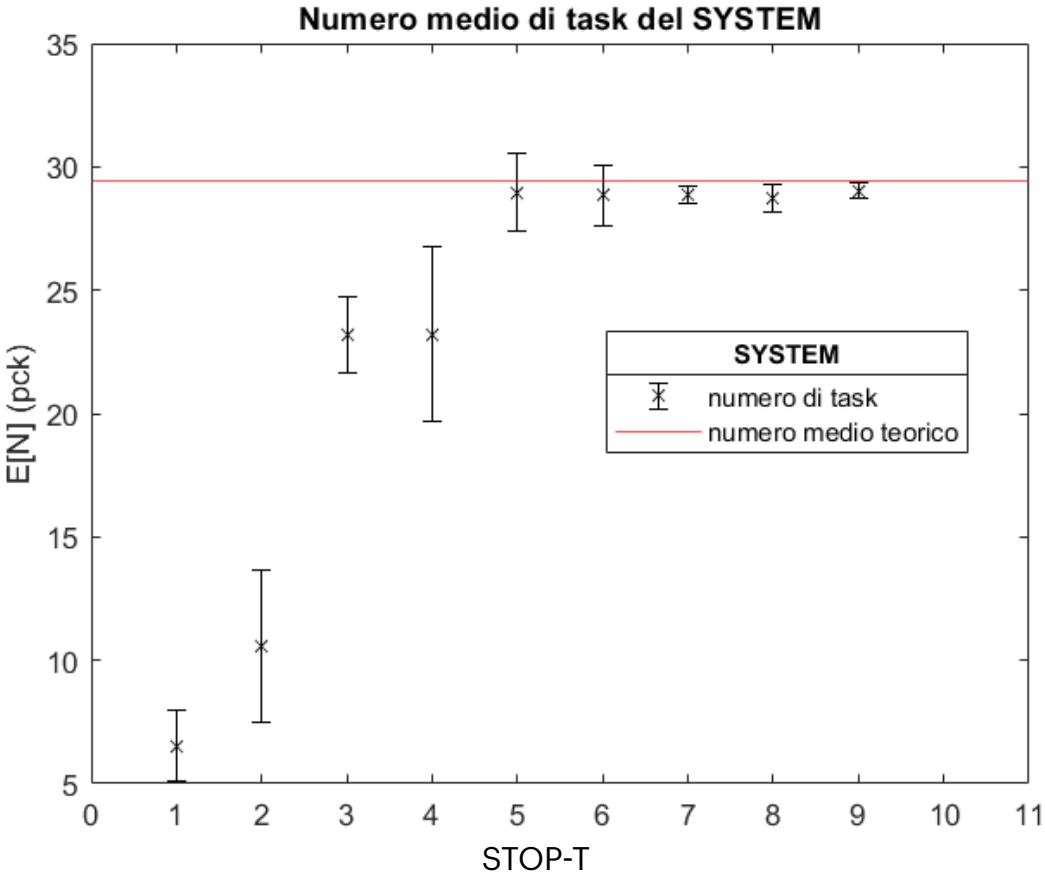
E per l’algoritmo 2:




STOP-T

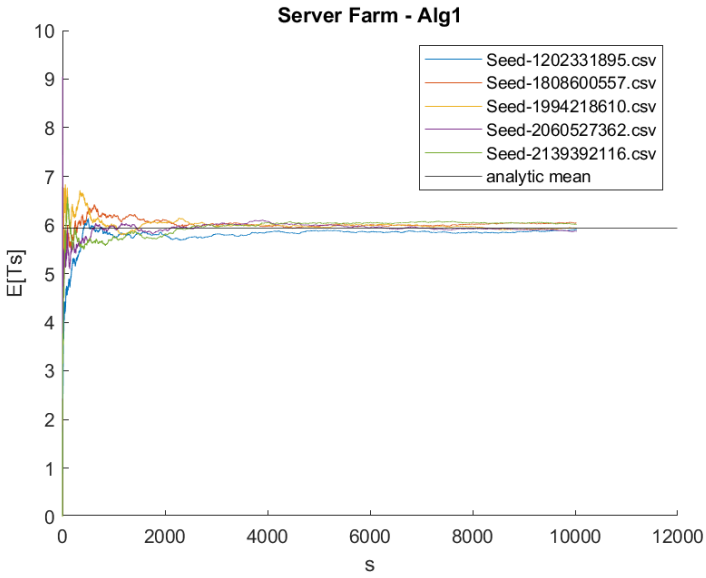


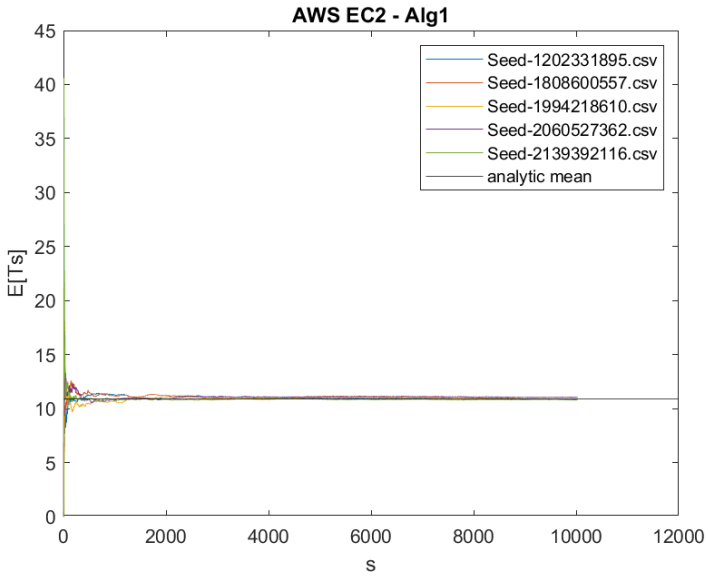

STOP-T

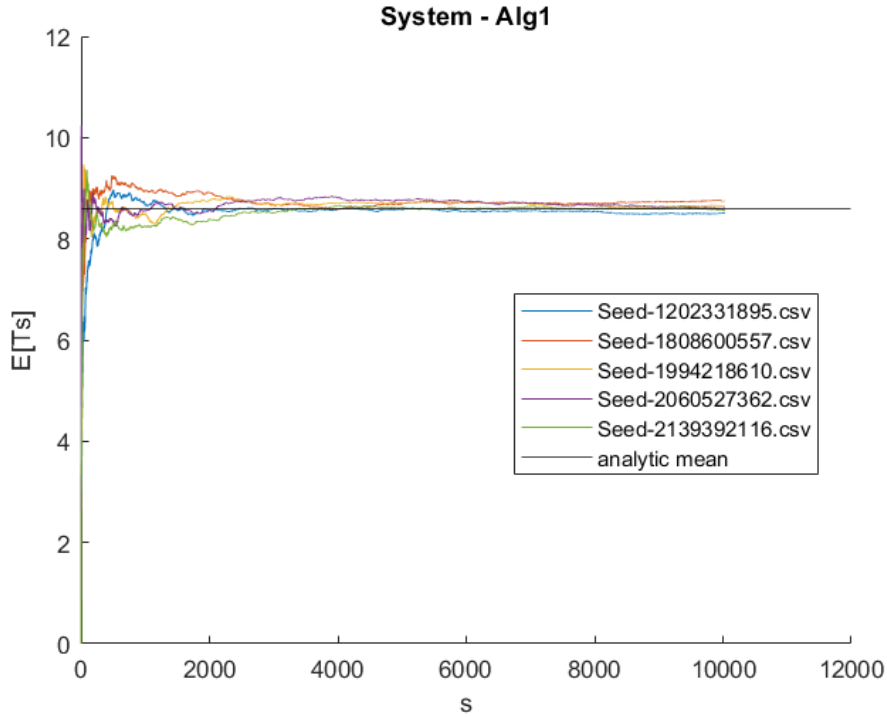


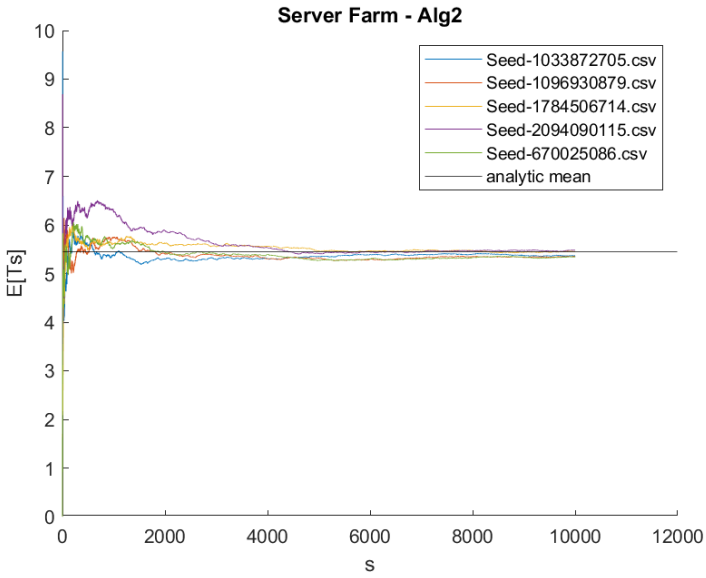

STOP-T

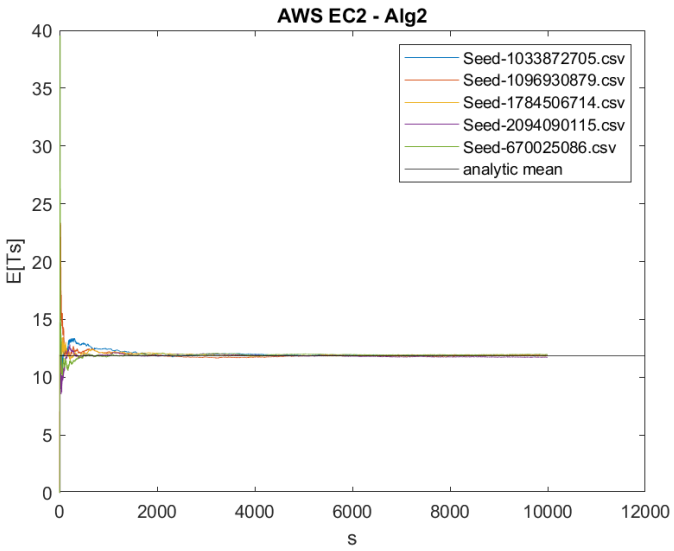
Possiamo adesso osservare gli andamenti del tempo di risposta istante per istante in funzione del valore del clock, per entrambi gli algoritmi nella fase transiente del sistema per evidenziarne la stazionarietà. Il valore del tempo di risposta, infatti, tende in modo evidente al corrispondente valore teorico, anche al variare dei seed utilizzati (con le modalità riportate in precedenza), riportati nella legenda.

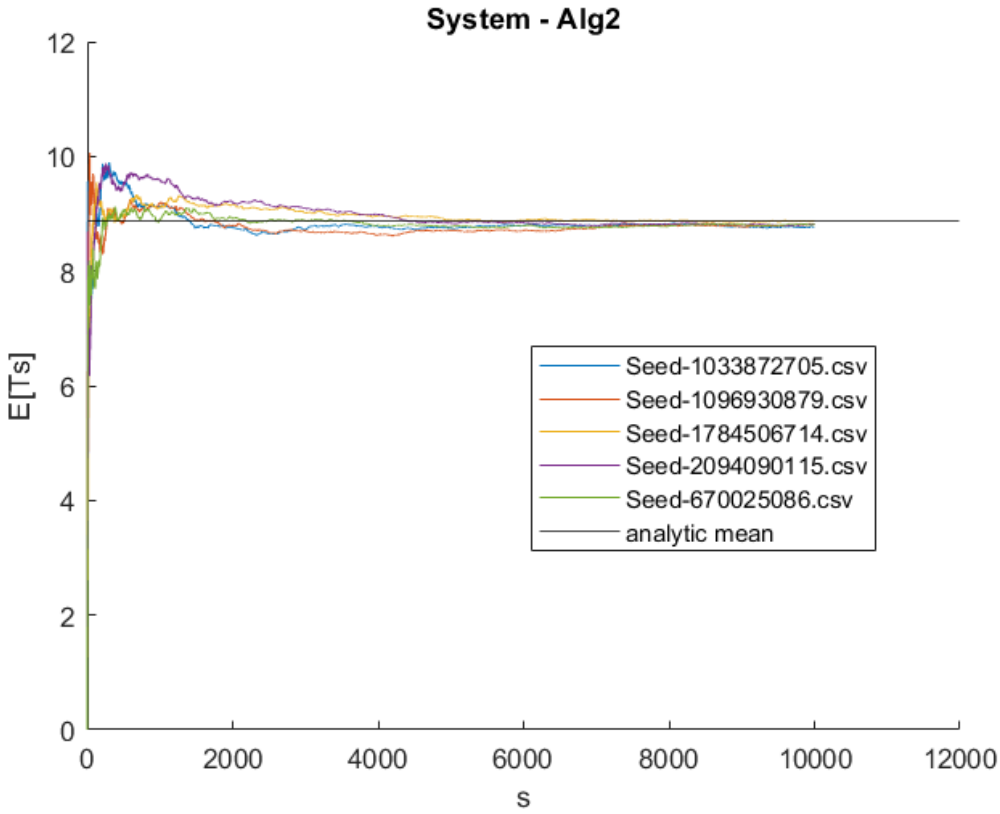



Server Farm - Alg2
Seed-1033872705.csv
Seed-1096930879.csv
Seed-1784506714.csv
Seed-2094090115.csv
Seed-670025086.csv
analytic mean
E[Ts]
s


AWS EC2 - Alg2
Seed-1033872705.csv
Seed-1096930879.csv
Seed-1784506714.csv
Seed-2094090115.csv
Seed-670025086.csv
analytic mean
E[Ts]
s


System - Alg2
Seed-1033872705.csv
Seed-1096930879.csv
Seed-1784506714.csv
Seed-2094090115.csv
Seed-670025086.csv
analytic mean
E[Ts]
s

# Conclusioni

Lo studio di questo tipo di problema ha evidenziato come un sistema distribuito multitask, come può essere un social network, sia particolarmente complicato da modellare: avere un flusso di richieste in arrivo che vengono trattate in modo diverso in base al tipo di job, aumenta notevolmente la difficoltà dell'analisi.

L'obiettivo principale era quello di migliorare, attraverso una nuova versione dell'algoritmo di assegnazione dei task del controllore, il tempo medio di risposta globale per le letture (task 1). Confrontando i valori riportati è possibile vedere come, grazie all'Algoritmo 2, per questo specifico caso l'attesa dell'utente sia effettivamente diminuita, a discapito delle scritture che vengono sempre servite dalle istanze di AWS EC2. In un social network, effettivamente, è abitudine attendere meno per le operazioni di lettura (descritte ad inizio trattazione) e maggiormente per le operazioni di scrittura (più gravose a livello computazionale, quindi ci va bene che vengano servite dalle istanze di AWS).

Per come abbiamo pensato il caso in esame, il numero di scritture ha una frequenza superiore, ma le letture sono agevolate all'interno della server farm, motivo per cui è stato deciso di accettare qualunque task fino ad una soglia LIMIT, per poi accettare nei server solo task di tipo 1, le letture appunto. Il miglioramento è evidente nel numero medio di task processati dalla Server Farm, ora non più sempre satura.

Il Server Farm, infatti, rappresenta un collo di bottiglia nel sistema, soprattutto se si considera la distribuzione iperesponenziale che descrive i suoi tempi di servizio: essendo caratterizzata da un'alta varianza, i valori dei tempi prodotti possono essere anche molto grandi, cosa che porta un utente con richiesta di lettura ad attendere già molto tempo ed è una limitazione avere un pool di server composto da risorse sempre

e solo non disponibili, soprattutto in un social network in cui l'attività media dell'utente è basata su visualizzazione di contenuti, quindi prevalentemente operazioni di lettura.

Sono state fatte delle simulazioni utilizzando un valore di LIMIT differente, per valutarne il comportamento. La scelta finale ha portato a LIMIT= 8 , perché il tempo medio di servizio delle istanze AWS EC2 per le scritture  resta comunque superiore rispetto a quello del Server Farm, aspetto che globalmente peggiora il tempo medio di risposta totale e che non può essere arginato a meno di miglioramenti a livello di hardware, che per una start-up agli albori abbiamo supposto non economicamente possibili.

Di seguito vengono riportate le motivazioni che ci hanno spinto a scegliere il valore di LIMIT sopra citato: con il valore 8 abbiamo i risultati migliori.

LIMIT = 7

tempo di risposta del server farm: 5.199001 s

tempo di risposta del server farm per task1 4.813578 s

tempo di risposta del server farm per task2 7.032679 s

tempo di risposta di AWS EC2: 12.037503 s

tempo di risposta di AWS EC2 per task1: 8.962027 s

tempo di risposta di AWS EC2 per task2: 12.487261 s

tempo medio di risposta del sistema: 8.898112 s

tempo di risposta sistema per task1: 5.200771 s

tempo di risposta sistema per task2: 12.238202 s

LIMIT = 8

tempo di risposta del server farm: 5.350781 s

tempo di risposta del server farm per task1: 4.810213 s

tempo di risposta del server farm per task2: 6.971103 s

tempo di risposta di AWS EC2: 11.630135 s

tempo di risposta di AWS EC2 per task1: 8.530002 s

tempo di risposta di AWS EC2 per task2: 12.251871 s

tempo medio di risposta del sistema: 8.662509 s

tempo di risposta sistema per task1: 5.089525 s

tempo di risposta sistema per task2: 12.057250 s

LIMIT = 9

tempo di risposta del server farm: 5.436640 s

tempo di risposta del server farm per task1: 5.064647 s

tempo di risposta del server farm per task2: 6.161386 s

tempo di risposta di AWS EC2: 11.558720 s

tempo di risposta di AWS EC2 per task1: 9.426542 s

tempo di risposta di AWS EC2 per task2: 12.306203 s

tempo medio di risposta del sistema: 8.863268 s

tempo di risposta sistema per task1: 6.680616 s

tempo di risposta sistema per task2: 11.044029 s

Come è possibile notare, avendo come LIMIT il valore 7, i task vengono gestiti maggiormente da AWS EC2 così da aumentarne il tempo di risposta. Un valore come 9 invece appesantisce troppo la Server Farm. Il trade off che minimizza il tempo di risposta globale e del sistema per task di tipo 1 è il valore 8.