Progetto di Simulazione di Sistemi

A queueing system with decomposed service and inventoried preliminary services

Indice

1. Introduzione

Il seguente lavoro ha l'obiettivo di analizzare un sistema single-server con una coda all'ingresso, dove il server è composto da due stage indipendenti: il primo è generico e può eseguire calcoli anche in assenza dell'utente (PS), mentre il secondo necessita della presenza dell'utente per essere portato a termine. Il primo stage (PS), in assenza di utente, può salvare un inventario di "primi stage" precalcolati per ridurre la permanenza dell'utente nel sistema.

Un sistema di questo tipo può essere usato nel mondo reale per diminuire il tempo di permanenza nel sistema nel caso in cui ci siano porzioni del servizio ripetitive e prevedibili.

2. Tool Utilizzati

La piattaforma utilizzata per la modellazione, la configurazione e la simulazione del sistema è **OMNeT++ versione 5.5.1**.
Per la fase di pulizia, organizzazione e analisi dei dati sono state usate le librerie **SciPy**, **NumPy** e **Pandas** per il linguaggio di programmazione **Python v. 3.7**.
Altro strumento usato per organizzazione dei dati e creazione di grafici è stato **GoogleSpreadSheet**

3. Modello

3.1. Descrizione

Il modello da implementare presuppone l'esistenza di due tipi di utenti **U1** e **U2** (chiamati Job) all'interno del sistema ed è composto da:

- Coda all'ingresso, di tipo FIFO (first-in first-out) e prioritaria che da priorità di servizio all'utente di tipo U1.
- Server multi-stage suddiviso in:
  - Preliminary Service (**PS**) il quale ha un tempo di servizio che segue la media della distribuzione esponenziale **1/β**.
    In assenza di richieste da utenti può pre-calcolare e salvare un numero ***n*** di generici "first-stage".
    La produzione di first-stage generici ha un tempo di servizio che segue la media della distribuzione esponenziale **1/α** e avviene seguendo una probabilità ***p*** in favore dell'utente di tipo U1 e una probabilità **1-p** in favore di un utente U2.
    Nel caso di richiesta durante la fase di pre-produzione, si interrompe e serve l'utente in ingresso
  - Complementary Service (**CS**) che si occupa della parte di Server non generica e più specifica (non pre-calcolabile) e segue la media della distribuzione esponenziale **1/γ**

3.2. Parametri

La simulazione sul modello è stata eseguita su diversi parametri:

- **n**: numero massimo di Preliminary Service pre-calcolati e immagazzinati nel sistema (per tipologia di utente)
- **m(U1)** e **m(U2)**: media della distribuzione esponenziale dei tempi di interarrivo per utenti di tipo U1 ed utenti di tipo U2

- **z(U1)** e **z(U2)**: media della distribuzione esponenziale del tempo di servizio (per utenti di tipo U1 ed U2) per il Preliminary Service (PS) nel caso in cui stia producendo PS in mancanza di utenti
- **w(U1)** e **w(U2)**: media della distribuzione esponenziale del tempo di servizio (per utenti di tipo U1 ed U2) per il Preliminary Service (PS) nel caso in cui sia presente un utente da servire direttamente (senza attingere all'inventario di PS già prodotti)
- **y(U1)** e **y(U2)**: media della distribuzione esponenziale del tempo di servizio (per utenti di tipo U1 ed U2) per il Complementary Service (CS)
- **p**: probabilità che la produzione di PS avvenga per utenti di tipo U1 (1-p che avvenga per utenti di tipo U2)

I valori assegnati a questi parametri sono:

- **n**: 2, 3, 4, 5
- **m(U1)**: 2s, 3s, 4s **m(U2)**: 4s, 5s, 6s
- **z(U1)**: 0.8s, 1.0s, 1.4s **z(U2)**: 1.0s, 1.2s, 1,4s
- **w(U1)**: 1.2s, 1.4s, 1.8s **w(U2)**: 1.4s, 1.6s, 1.8s
- **y(U1)**: 0.8s, 1.4s, 2,3s **y(U2)**: 1.0s, 1.8s, 2,3s
- **p**: 0.5, 0.7, 0.8
-

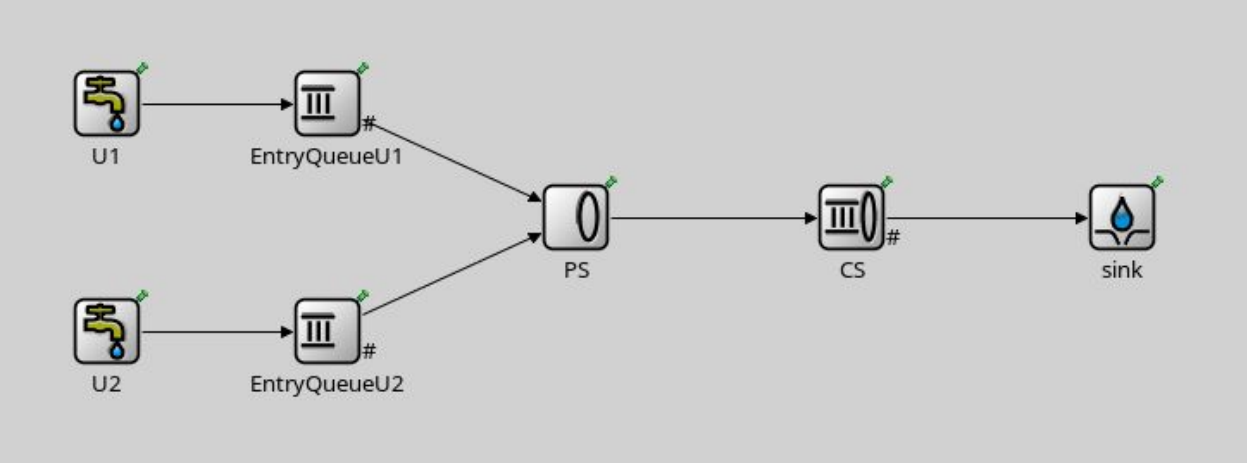
Vi è un totale di 12 configurazioni, identificate nel file *omnet.ini* con {*config*}-n{*n*}.

Config (First, Second, Third) identifica una delle 3 configurazioni di valori principali composte dall'i-esimo valore per ogni parametro, mentre *n* il valore di n che viene variato per ogni configurazione.

E.g: First-n2 è la configurazione composta da il valore 1 di ogni parametro ed n=2.

4. Implementazione

Per l'implementazione sono stati utilizzati i moduli implementati dalla libreria *queuinglib*, apportando le opportune modifiche ai nodi *Source*, *PassiveQueue*, *Server*, *Queue*, *Sink.*



4.1. Source

Il modulo Source si occupa di generare i job ad ogni *interArrivalTime*. Dopo aver generato un Job, viene inserito in una coda all'ingresso del sistema in attesa di essere servito.

4.2. PassiveQueue(s)

Il modulo *PassiveQueue* si occupa di mantenere in coda i Job generati dai Source. Ne vengono usate due per facilitare il modulo *Server* nel capire la tipologia di utente per gestire la priorità.

4.3. Server (PS)

Il modulo *Server* si occupa di simulare un tempo di permanenza al proprio interno definito dal parametro *serviceTime.* Tale modulo è stato modificato per occuparsi, in più, di due casistiche:

- gestire la preproduzione di PS per ogni utente nel caso non ci siano utenti da servire
- servire gli utenti pescando dall'inventario di PS o, nel caso non ci siano PS già pre-calcolati, servirli direttamente.

Tale modulo non pesca dalla coda all'ingresso finchè nel sistema (PS + CS) vi è presente un utente (job)

Nell'*handleMessage()* vengono aggiunti:

- Produzione di PS probabilistica in base al tipo di utente con scheduling di un msg *endProducingMsg* per ogni tipologia di utente

```
if(bernoulli(p)){
  if(inventoriedPSU1 < n) {
    ...
    scheduleAt(simTime()+serviceTimeForInventory,endProducingMsgU1);
  }
} else {
  if(inventoriedPSU2 < n) {
    ...
    scheduleAt(simTime()+serviceTimeForInventory,endProducingMsgU2);
  }
}
```

- due controlli per catturare questi *endProducingMsgU1* ed *endProducingMsgU2* e riavviare la produzione

- Nel caso non sia un msg di endServiceMsg o endProducing, il cancel di eventuali produzioni in corso e il servizio del Job

```
cancelEvent(endProducingMsgU1);
cancelEvent(endProducingMsgU2);
...
if (!allocated)
  error("PS already allocated");
...
jobServiced = check_and_cast<Job *>(msg);
```

Viene implementato anche un nuovo metodo *deallocate(),* per deallocare il modulo PS da un modulo esterno (CS in questo caso)

```
void Server::deallocate()
{
   allocated = false;
   ...
   int k = selectionStrategy->select();
   ...
   if (k >= 0) {
      cGate *gate = selectionStrategy->selectableGate(k);
      check_and_cast<IPassiveQueue*>(
        gate->getOwnerModule()
        )->request(gate->getIndex());
   }
}
```

4.4. Queue (CS)

Il modulo *Queue* si occupa di implementare di base un server con coda all'ingresso. Il tempo di permanenza è definito dal parametro *serviceTime.* L'unica aggiunta è stata fatta al metodo *endService(...)* per deallocare PS una volta servito il Job.

```
...
((Server*) job->getSenderModule())->deallocate();
```

...

4.5. Sink

Il modulo *Sink* si occupa di distruggere i Job usciti dal sistema e raccogliere le statistiche sulla loro permanenza nel sistema utili all'analisi dei dati prodotti dalle simulazioni eseguite

5. Statistica

L'obiettivo era quello di simulare il sistema e calcolare quanto effettivamente, per ogni configurazione, migliorasse il tempo di permanenza all'interno del sistema all'aumentare del numero dei processi PS pre-calcolati e immagazzinati.

Per ogni combinazione di *configurazioni* e valori di *n* effettuata sono stati calcolati media con relativi intervalli di confidenza di:

- Lunghezza media delle coda all'ingresso del sistema (anche per singolo utente)
- Tempo di Soggiorno medio all'interno del sistema
- Tempo di Permanenza massima e minima all'interno del sistema

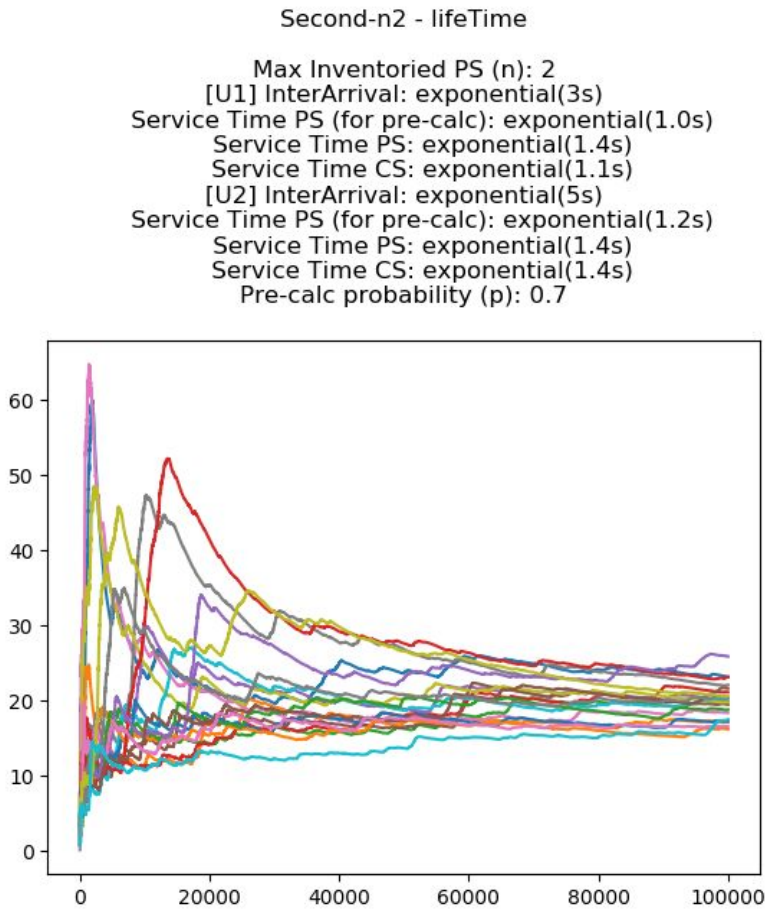
Ogni combinazione (12) è stata ripetuta 20 volte, per un tempo complessivo di 100000s.

5.1 Transiente Iniziale

Nel momento in cui un sistema inizia da poco ad essere operativo, tale sistema è fortemente influenzato dallo stato iniziale e dal tempo

trascorso prima dell'attivazione. Questa condizione del sistema è detta *transitoria.* Trascorso un lasso di tempo iniziale il sistema si stabilizza e raggiunge condizioni di equilibrio o stazionarie.

Durante gli esperimenti visualizzando i risultati ottenuti è stato notato un comportamento anomalo nella fase iniziale. Viene mostrata una configurazione d'esempio per questo comportamento:



È evidente che nelle prime fasi di simulazione il sistema è instabile. Valutando tutte le configurazioni si è deciso di eliminare questa fase e analizzare il sistema da quando va a regime. La fase transiente iniziale viene rimossa grazie alla funzionalità *wake-up time* di OMNeT++. La fase transiente individuata per questo sistema è di

30000 unità di tempo simulato, su un totale di 100000 unità di tempo totale della simulazione.

Grafici relativi al comportamento del sistema con o senza la fase di transiente iniziale possono essere trovato nelle cartelle:

- *charts/charts_vector_results*
- *charts/charts_vector_results_no_transient*

5.2 Risultati

Innanzitutto, prima di procedere con l'analisi, va evidenziato come ogni valore relativo all'utente di tipo 2 (lifeTime e queueLength) in 6 configurazioni su 12 non si stabilizza attorno ad un valore, ma continua a crescere.

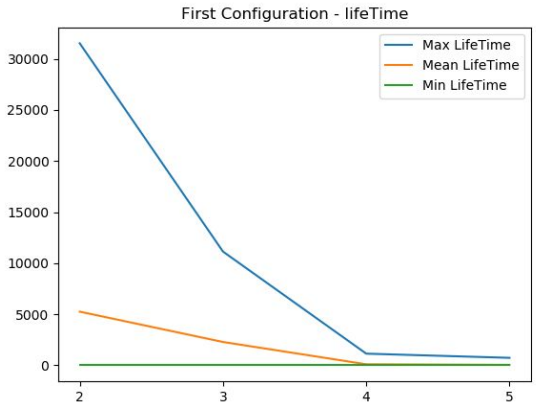
Tale comportamento è stato imputato a:

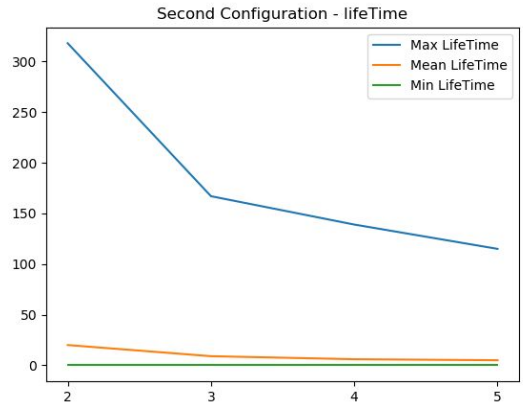
- presenza di una coda prioritaria per utenti di tipo 1 all'ingresso del sistema
- generazione di job con tempi di interarrivo di U1 inferiori a quelli di U2.

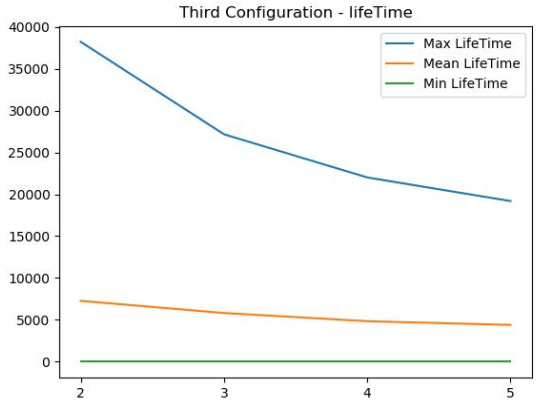
Questa peculiarità del sistema preso in esame, va ad inficiare anche sulle misurazioni del sistema complessivo (non separate per tipologia di utenti).

Dato importante da evidenziare è che all'aumentare di *n* (numero di PS pre-calcolati) il lifeTime medio (tempo di permanenza) dei job nel sistema cala drasticamente anche per le configurazioni che mediamente hanno lifeTime medi elevati.

Di seguito vengono riportati i grafici del *lifeTime* massimo, medio e minimo per visualizzare questo comportamento del sistema:



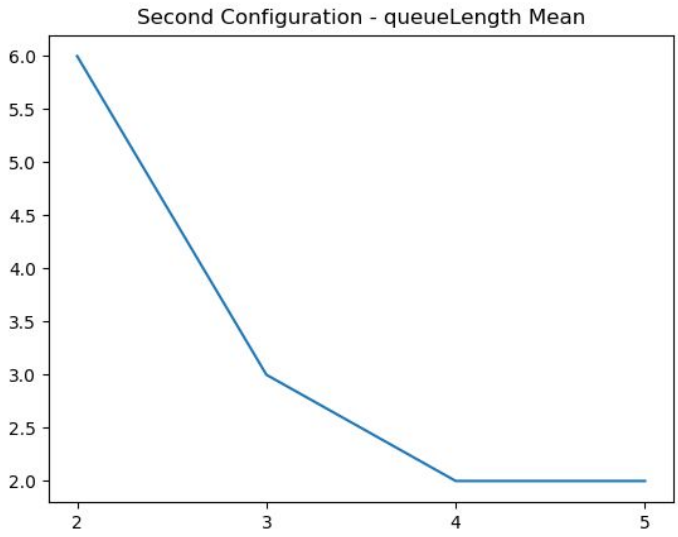




Risulta evidente che, all'aumentare del numero di PS pre-calcolati, il sistema migliora la propria responsività e riesce a servire i Job al proprio interno in modo molto più veloce. Questo conferma l'utilità dell'approccio adottato dal paper, anche complicando il caso di studio con diverse tipologie di utenti con differenti priorità.

La diminuzione del lifeTime massimo, medio e minimo si verifica anche per i singoli utenti, da ciò possiamo capire che i miglioramenti medi del sistema non è singolare di un tipo di utenti, ma complessivo per qualsiasi job debba essere servito dal sistema simulato.

L’aumento dei PS pre-calcolati ha un impatto positivo anche nella riduzione della lunghezza della coda all’ingresso del sistema, sia complessivamente che per singolo utente.



5.3 Intervalli di Confidenza

Partendo dai dati scalari di ogni configurazione registrati tramite OMNeT++, sono stati estratti i lifeTime e calcolati gli intervalli di confidenza per ogni misurazione richiesta.

I passi implementati per calcolare gli intervalli di confidenza sono i seguenti:

1. Calcolo Media delle medie di ogni run (20)
2. Calcolo Varianza (**var**)
3. Calcolo Deviazione Standard (**std_dev**)
4. Calcolo Errore standard della media

Successivamente è stato calcolato il valore di t di Student, con 19 gradi di libertà (20-1 = 19) e le percentuali (90% e 95%), attraverso la seguente formula: **Z (a/2) * (std_dev * √var /√n)**

Posto che :

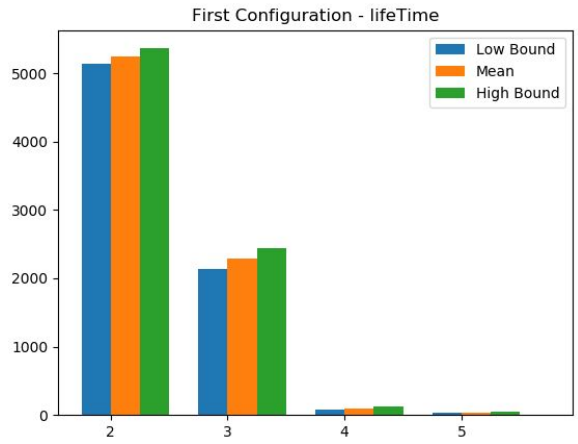
- a = 1 - 0.90 = 0.10 => Z(a/2) = 0.050 per 90% (0.90)
- a = 1 - 0.95 = 0.05 => Z(a/2) = 0.025 per 95% (0.95)

***t* Table**

cum. prob	$t_{.50}$	$t_{.75}$	$t_{.80}$	$t_{.85}$	$t_{.90}$	$t_{.95}$	$t_{.975}$	$t_{.99}$	$t_{.995}$	$t_{.999}$	$t_{.9995}$
one-tail	0.50	0.25	0.20	0.15	0.10	0.05	0.025	0.01	0.005	0.001	0.0005
two-tails	1.00	0.50	0.40	0.30	0.20	0.10	0.05	0.02	0.01	0.002	0.001
df											
19	0.000	0.688	0.861	1.066	1.328	1.729	2.093	2.539	2.861	3.579	3.883

Per il calcolo degli intervalli di confidenza (limite superiore e inferiore) per t95 e t90:

- Min_90 = media - (t90 * std_dev * √var /√n)
- Max_90 = media + (t90 * std_dev * √var /√n)
- Min_95 = media - (t95 * std_dev * √var /√n)
- Max_95 = media + (t95 * std_dev * √var /√n)



5.3 Plotting e processing dei Dati

Tutti i dati prodotti da OMNeT++ sono stati salvati su file di tipo *.vec, *.sca e *.vci.

Tramite *scavetool* sono stati estratti sia i vettori di tutte le osservazione simulate, sia i valori scalari generati dal sistema. Gli script utilizzati per l'estrazione possono esser trovati alla path *./StatisticalAnalysis/create_*.sh*

I file generati sono stati analizzati successivamente tramite degli script in python che possono essere trovati alla path *./StatisticalAnalysis/plot_*.py* e *./StatisticalAnalysis/clean_json.py*. Lo script clean_json.py si occupa di fare una pulizia dei file esportati per alleggerire i file da processare per organizzare i dati nei csv riassuntivi e per generare i plot. I tre script che si occupano dei plot invece, si importano i dati dai file volta per volta e generano i grafici necessari all'analisi statistica.

6. Riferimenti

1. A queueing system with decomposed service and inventoried preliminary services (Gabi Hanukova, Tal Avinadava, Tatyana Chernonoga , Uriel Spiegel, Uri Yechiali)
2. OMNeT++ [https://omnetpp.org/intro]
3. Python [https://www.python.org/]