RightSizing-SpikeServer

1 Caso di Studio

Il sistema oggetto di studio è un'architettura di data center per un Internet Service Provider, progettata per gestire dinamicamente le fluttuazioni di carico e garantire la Quality of Service (QoS) ottimizzando al contempo l'uso delle risorse.

Il problema principale affrontato è il "right-sizing", ovvero come evitare sia il sovradimensionamento (spreco di risorse) sia il sottodimensionamento (violazione degli SLA e degrado delle prestazioni), specialmente in presenza di fluttuazioni di carico a breve e lungo termine.

L'architettura proposta, come descritto nel caso di studio 6.2 del libro di testo "Performance Engineer", si basa su un livello di scaling verticale che gestisce i picchi di carico improvvisi e di breve durata. Questo livello introduce uno Spike Server dedicato. Un Load Controller monitora un indicatore di picco (Spike Indicator, SI), definito come il numero di richieste concorrenti totali presenti su un Web Server (tutte le richieste vengono gestite con un scheduler processor sharing).

Il comportamento del sistema seguirebbe quanto descritto:

- Quando l'indicatore SI supera una soglia di allarme SI_max, le nuove richieste in arrivo non vengono più inviate al Web Server congestionato, ma vengono reindirizzate allo Spike Server.
- Quando il carico sul Web Server diminuisce e SI scende al di sotto della soglia, il routing delle richieste torna alla normalità.

2 Obiettivi dello studio

Lo studio si pone l'obiettivo di analizzare e validare l'efficacia del modello di autoscaling basato su spike server attraverso la simulazione. Gli obiettivi specifici sono:

- Determinare il valore di SI_max più alto possibile (provando vari valori di SI) che mantenga comunque il tempo di risposta medio $E[R] \leq 8$ secondi, con un tasso di arrivo iperesponenziale con media di 6.66 req/s (400 req/min).

- Analizzare come varia il tempo di risposta medio al variare del coefficiente di variazione degli arrivi iperesponenziali (fluttuazioni a breve termine) per carichi di lavoro crescenti (da 1 req/s a 12 req/s).
- Verificare come cambia il contesto (sempre sotto carichi crescenti e al variare di SI_max) se lo spike server ha un tasso di servizio doppio rispetto al webserver principale, invece che uguale come nel caso base.

3 Modello concettuale

Il modello descritto può essere schematizzato come in figura 1.

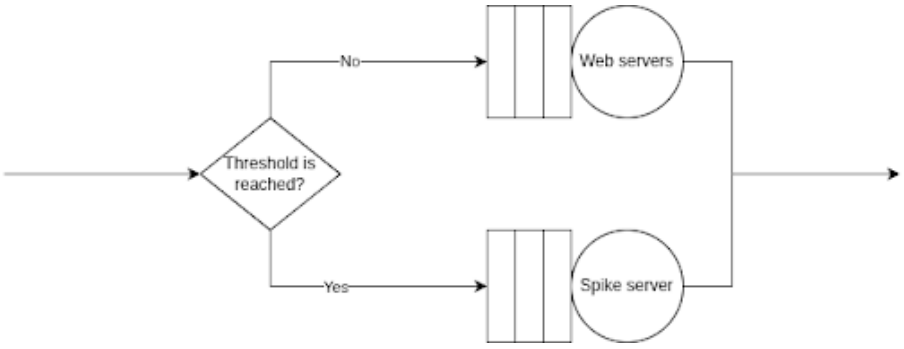

Figure 1: Modello concettuale del sistema di autoscaling gerarchico con Spike Server

I job arrivano e a seconda del livello di pienezza dei webservers. Nonostante il sistema possa sembrare troppo semplice per una analisi simulativa, il fatto che il routing non sia probalistico lo rende molto complesso da analizzare in modo analitico. il routing dei job, dipende strettamente dallo stato del webserver nel momento del routing.

Spiegazione tempi di servizio esponenziale: Nel caso di studio affrontato nel libro viene utilizzata una distribuzione iperesponenziale nel tassi di servizio per modellare il fatto che ad un server arrivano job di dimensione molto variabile. Ovviamente questa cosa potrebbe essere modellata anche utilizzando delle size dei job differenti anziché agire sul tasso di servizio. Tuttavia ho deciso di attenermi al testo originale e utilizzare anche io dei tassi di servizio iperesponenziali per modellare questo comportamento per poter confrontare meglio i risultati ottenuti.

3.1 Spike server

Lo Spike Server avrà la stessa potenza rispetto al webserver nei primi 2 obiettivi. Nell'obiettivo 3, invece, avrà un tasso di servizio doppio rispetto a quello del webserver principale, in modo da poter smaltire rapidamente i picchi di carico. Per il calcolo di SI_max ottimle è importante sottolineare che lo spike server

è bene usarlo il meno possibile, in quanto è una risorsa più costosa rispetto al webserver. Quindi anziché cercare di minimizzare il tempo di risposta medio del sistema, si cerca di rispettare lo SLA (8 secondi) utilizzando il meno possibile lo spike server.

3.2 Web Server

Il web server sarà solamente uno, per semplicità di modellazione. Per questo motivo non vengono gestite le fluttuazioni a lungo termine, in quanto queste richiederebbero un sistema con più webserver per utilizzare una scaling orizzontale.

4 Modello delle specifiche

Queste variabili matematiche, sono la rappresentazione dello stato del sistema:

- $SI(t) \in [0, SI_{max})$: Spike indicator, indica il numero di job presenti nel web server al tempo t.
- $n_{spike}(t) \in [0, \infty)$: numero di job presenti nello spike server al tempo t.

per cui lo stato del sistema è definito come la tupla $S(t) = (SI(t), n_{spike}(t))$.

4.1 Componenti fisici del sistema:

- Web server principale: server principale che gestisce le richieste in arrivo.
- Spike server: server secondario che viene attivato quando il carico sul web server principale supera una certa soglia (SI_{max}).

Componenti logici del sistema:

- Load Controller: componente che monitora l'indicatore di picco (Spike Indicator, SI) e decide il routing delle richieste in base al valore di SI e alla soglia SI_{max}.

4.2 Specifiche del carico di lavoro:

- Arrivi:
  - Valore base: Processo di arrivo iperesponenziale con cv = 4 e media 0.15 secondi (6.66 req/s)
  - Valore stress: Processo di arrivo iperesponenziale con carico variabile da 1 req/s a 12 req/s
- Servizio: Distribuzione iperesponenziale con cv = 4 e media 0.16 secondi (web server e spike server), media 0.08 secondi (spike server nell'obiettivo 3)

- Scheduling: Processor Sharing

Vale la pena notare che, nel caso base del webserver, siccome $\rho > 1$, il sistema è instabile e senza spike server il tempo di risposta divergerebbe a infinito.

4.3 Logica di controllo:

- Arrivi nuovi job:
  - Se $SI < SI_{max}$: invia al webserver principale
  - Se $SI \geq SI_{max}$: invia allo spike server
- Completamento job:
  - Decrementa il contatore SI se il job era nel webserver principale
  - Decrementa il contatore n_spike se il job era nello spike server

Nella logica originale del caso di studio, il valore di SI parte da SI_{max} e viene decrementato ad ogni completamento di un job. La logica da me adottata, nonostante sia inversa, è perfettamente equivalente.

4.4 Metrica di valutazione delle prestazioni:

Lo SLA da rispettare è un tempo di risposta medio $E[R] \leq 8$ secondi.

4.5 Gestione della fine della simulazione:

Dato che il sistema viene studiato a regime, esso fa riferimento ad un orizzonte temporale prefissato che va dai 120 secondi (per eliminare il bias della fase transitoria in cui le code sono ancora vuote) ai 1200 secondi di simulazione. Questo viene fatto nel report di riferimento e quindi per confrontare al meglio i risultati si è deciso di adottare lo stesso approccio. Inoltre, non verrà fatta allo scadere del tempo di simulazione una pulizia delle code, ma si considereranno solo i job completati entro il tempo di simulazione, per il calcolo delle metriche medie aggregate (es. R0), coerentemente con l'approccio adottato nel caso di studio di riferimento.

5 Modello Computazionale

Il sistema di simulazione è implementato in Python e segue un modello ad eventi discreti. Sono state utilizzate alcuni file della libreria di Steve Park & Dave Geyer, tradotti in python da Philip Steele. In particolare sono state utilizzate le librerie per la generazione di numeri casuali per il multistream (rngs.py) e per le distribuzioni (rvgs.py).

5.1 Stato del sistema

Le variabili di programmazione che rappresentano lo stato del sistema sono:

- web_jobs : list: è una lista che contiene i job in fase di processamento nel web server, la sua lunghezza rappresenta l'indicatore SI
- spike_jobs : list: è una lista che contiene i job in fase di processamento nello spike server.

5.2 Descrizione del job

Ogni job è rappresentato da una classe Job:

```
class Job:
    def __init__(self,
                 arrival_time,
                 service_demand,
                 is_spike=False):
        self.arrival_time = arrival_time
        self.service_demand = service_demand
        self.remaining_work = service_demand
        self.is_spike = is_spike
```

Dove:

- arrival_time: tempo di arrivo del job
- service_demand: tempo di servizio totale richiesto dal job
- remaining_work: lavoro rimanente da completare (utile per lo scheduling processor sharing)
- is_spike: booleano che indica se il job è stato assegnato allo spike server

5.3 Identificazione degli eventi

- arrivo di un nuovo job
- completamento di un job nel web server
- completamento di un job nello spike server

5.4 Logica di controllo e routing

Ogni volta che arriva un nuovo job:

- Se len(web_jobs) $\leq SI_{max}$, il job viene aggiunto a web_jobs.
- Se len(web_jobs) $> SI_{max}$, il job viene aggiunto a spike_jobs.

Ogni volta che job completa rimuove dalla lista corrispondente il job (web_jobs o spike_jobs).

5.5 Implementazione dello scheduler

Al contrario di quanto avviene nel caso di studio di riferimento in cui si usa uno scheduling processor sharing.

5.6 Configurazione dei parametri delle distribuzioni

- Stream 0: Inter-arrivi (Iperesponenziale, media 0.15s, cv=4).
- Stream 1: Servizio Web Server (Iperesponenziale, media 0.16s, cv=4).
- Stream 2: Servizio Spike Server (Iperesponenziale, media 0.08s, cv=4).

La distribuzione iperesponenziale è stata implementata utilizzando l'esponenziale e l'uniforme della libreria rvgs.py. Con l'uniforme calcoliamo la probabilità di scegliere una delle due esponenziali, e con l'esponenziale calcoliamo il servizio vero e proprio (Figura 2).

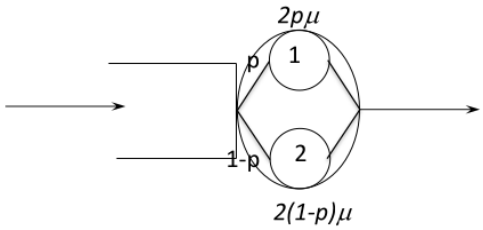

Figure 2: Generazione di una variabile casuale iperesponenziale

In particolare date in input la media e il coefficiente di variazione cv. Sapendo che $g(p) = cv^2$, possiamo calcolare p_1, p_2, m_1 e m_2 come:

$$c = \frac{cv^2 - 1}{cv^2 + 1}$$

$$p_1 = \frac{1 + c}{2}$$

$$p_2 = 1 - p_1$$

$$m_1 = \frac{mean}{2 \cdot p_1}$$

$$m_2 = \frac{mean}{2 \cdot p_2}$$

e quindi utilizzare un semplice if per scegliere quale esponenziale utilizzare in base ad un campione uniforme:

```
u = Uniform(0,1)
if u < p1:
    return Exponential(m1)
else:
    return Exponential(m2)
```

Da notare che per ogni campionamento dell'esponenziale si utilizzano, quindi, due estrazioni dallo stream del generatore di numeri casuali.

5.7 Struttura del simulatore

Il simulatore è strutturato come una classe Simulator che gestisce l'inizializzazione, l'esecuzione e la raccolta delle metriche della simulazione. La classe ha una serie di costanti di configurazione:

```
START      = 0.0
BIAS_PHASE = 120.0
STOP       = 1200.0
INFINITY   = 1e15
SEED       = 8
REPLICAS   = 100
N_PROCESSES = 12
```

Dove:

- START: tempo di inizio della simulazione
- BIAS_PHASE: durata della fase transitoria (120 secondi)
- STOP: tempo di fine della simulazione (1200 secondi)
- INFINITY: un valore molto grande usato per inizializzare i tempi di evento (deve essere molto più grande di STOP)
- SEED: seme per il generatore di numeri casuali, che è stato utilizzato insieme alla libreria rngs.py per tutti i generatori di numeri casuali
- REPLICAS: numero di repliche della simulazione (100 repliche)
- N_PROCESSES: numero di processi paralleli per l'esecuzione delle repliche (12 processi)

5.7.1 Parallelismo

Per velocizzare l'esecuzione delle repliche della simulazione, è stata implementata una logica di multiprocessing per sfruttare tutti i core della mia CPU disponibili. Tuttavia per farlo ho dovuto prendere alcune accortezze che permettessero di non minare nè l'indipendenza delle repliche nè la riproducibilità dei risultati. In particolare:

- ad ogni processo vengono assegnati 3 stream dei 256 disponibili, in modo che ogni processo abbia il proprio spazio di numeri casuali indipendenti dagli altri processi, ma in maniera tale che potessero procedere in parallelo senza che la non deterministicità del parallelismo influisse sull'ordine di estrazione dei numeri casuali.

- Per il numero di repliche introdotte, ogni processo non si avvicinava nemmeno lontanamente al limite dello stream assegnato.

  $$MODULUS/STREAM \approx 8388607$$

  Ho comunque implementato un meccanismo in cui una volta sforato il limite dello stream, gli venivano assegnati 3 nuovi stream. Ed essendo il numero di processi 12, e ogni processo usa 3 stream, questo è tranquillamente possibile senza sforare il limite dei 256 stream e senza ulteriori meccanismi di controllo.

- Sempre per garantire la riproducibilità dei risultati, ad ogni processo vengono assegnate sempre lo stesso numero di repliche da eseguire, in modo tale che le ultime repliche non vengano eseguite da processi diversi (e quindi strem diversi) in base all'assegnazione dinamica di essi.

5.8 Raccolta delle metriche

Per raccogliere le metriche è stata utilizzata una classe Track:

```
class Track:
    def __init__(self):
        self.area_node_web = 0.0
        self.area_node_spike = 0.0

        self.area_busy_web = 0.0
        self.area_busy_spike = 0.0

        self.completed_web = 0
        self.completed_spike = 0

        self.scaling_actions = 0
```

Dove:

- area_node_web: area sotto la curva del numero di job nel web server

  $$\int_{BIAS\_PHASE}^{STOP} N_{web}(t)dt$$

- area_node_spike: area sotto la curva del numero di job nello spike server

  $$\int_{BIAS\_PHASE}^{STOP} N_{spike}(t)dt$$

- area_busy_web: area sotto la curva del tempo di utilizzo del web server

$$\int_{BIAS\_PHASE}^{STOP} U_{web}(t)dt$$

- area_busy_spike: area sotto la curva del tempo di utilizzo dello spike server

$$\int_{BIAS\_PHASE}^{STOP} U_{spike}(t)dt$$

- completed_web: numero di job completati nel web server
- completed_spike: numero di job completati nello spike server
- scaling_actions: numero di azioni di scaling (numero di volte in cui lo spike server è stato attivato)

Gli integrali sono calcolati ad ogni evento in maniera incrementale durante la simulazione.

Le metriche finali raccolte sono state:

- **web_response_time**: tempo di risposta medio nel web server calcolato come l'integrale del numero di job completati nel web server diviso per il numero di job completati nel web server

$$\frac{area\_node\_web}{completed\_web}$$

- **spike_response_time**: tempo di risposta medio nello spike server

$$\frac{area\_node\_spike}{completed\_spike}$$

- **total_response_time**: tempo di risposta medio totale

$$\frac{area\_node\_web + area\_node\_spike}{completed\_web + completed\_spike}$$

- **utilization_web**: utilizzo medio del web server

$$\frac{area\_busy\_web}{STOP - BIAS\_PHASE}$$

- **utilization_spike**: utilizzo medio dello spike server

$$\frac{area\_busy\_spike}{STOP - BIAS\_PHASE}$$

- **throughput_web**: throughput medio del web server

$$\frac{completed\_web}{STOP - BIAS\_PHASE}$$

- **throughput_spike**: throughput medio dello spike server

$$\frac{completed\_spike}{STOP - BIAS\_PHASE}$$

- **throughput_total**: throughput medio totale

$$\frac{completed\_web + completed\_spike}{STOP - BIAS\_PHASE}$$

- **scaling_actions**: numero di azioni di scaling

Per quanto riguarda le metriche finali su tutte le repliche, è stato utilizzato l'algoritmo di Welford per il calcolo delle medie e varianze e poi degli intervalli di confidenza al 95%.

5.9 Scheduler

Siccome lo scheduler è di tipo processor sharing, il tempo prima del prossimo evento da schedulare è calcolato come:

$$min\{next\_arrival, time\_to\_complete\_web\_job, time\_to\_complete\_spike\_job\}$$

Dove:

- next_arrival: tempo rimanente al prossimo arrivo
- time_to_complete_web_job: tempo rimanente al completamento del job con il minor lavoro rimanente nel web server
- time_to_complete_spike_job: tempo rimanente al completamento del job con il minor lavoro rimanente nello spike server

Per calcolare i tempi rimanenti al completamento dei job, non basta trovare il job con il minor lavoro rimanente, perché essendo in processor sharing il tempo di completamento dipende anche dal numero di job in servizio. Quindi il tempo di completamento del job con il minor lavoro rimanente è calcolato come:

$$time\_to\_complete\_web\_job = min\_remaining\_work \times len(web\_jobs)$$

$$time\_to\_complete\_spike\_job = min\_remaining\_work \times len(spike\_jobs)$$

Una volta trovato il tempo del prossimo evento, si avanza il tempo di simulazione di tale intervallo e si aggiorna il lavoro rimanente di tutti i job in servizio sottraendo il lavoro svolto in tale intervallo:

$$j.remaining\_work = j.remaining\_work - \frac{time\_advance}{len(web\_jobs)} \forall j \in web\_jobs$$

$$j.remaining\_work = j.remaining\_work - \frac{time\_advance}{len(spike\_jobs)} \forall j \in spike\_jobs$$