Università degli Studi di Roma
Performance Modeling of Computer Systems and Networks

Corso di Laurea Magistrale in Ing. informatica

Relazione di progetto

Simulazione di Fluttuazioni nel Carico in sistemi con Autoscaling

Data relazione

Gruppo di lavoro:
Giuseppe Marseglia, giuseppe.marseglia@students.uniroma2.it
Matricola 0350066

Emanuele Gentili, emanuele.gentili.09@students.uniroma2.it
Matricola 0350087

Alessandro Finocchi, alessandro.finocchi@students.uniroma2.eu
Matricola 0340543

Docente:
Prof.ssa Vittoria Isabella De Nitto Personè

Anno Accademico 2024-2025

Indice

1 Introduzione

1.1 Contesto

Molti data center degli ISP soffrono di fluttuazioni nel carico dovute agli effetti combinati della variabilità sia del traffico in arrivo che del tempo computazionale delle richieste. Queste fluttuazioni possono essere a breve termine ("short-term") o a lungo termine ("long-term"), e comportano uno scenario variabile, in cui il problema di trovare il minimo numero di risorse da allocare per soddisfare gli obiettivi di performance non è banale: infatti fare over-provisioning potrebbe risultare uno spreco di risorse, mentre dall'altro lato l'under-provisioning può comportare violazioni in termini di QoS dal punto di vista del cliente.

Per risolverlo sono sempre più usate tecniche di **autoscaling**, le quali monitorano uno o più indicatori di performance, e al raggiungimento di una soglia adattano il numero di risorse.

Scalare orizzontalmente fornisce buoni risultati quando i carichi sono soggetti a fluttuazioni long-term, ma ha un impatto negativo in caso di fluttuazioni short-term dal momento che sono operazioni costose e lunghe. In particolare, gli improvvisi picchi di carico dovuti alle fluttuazioni short-term possono provocare periodi di congestione delle risorse improvvisi, con conseguente aumento dei tempi di risposta, e causare pericolose oscillazioni nel numero di risorse fornite, incoraggiando l'autoscaler orizzontale a prendere decisioni sbagliate nel breve termine.

Queste problematiche devono essere evitate quanto possibile dal momento che l'allocazione delle risorse ha un costo ed è time-consuming, ma possono essere risolte con l'ausilio di ulteriori risorse che permettono di ridurre la congestione sui server in maniera più veloce. Queste risorse non vengono aggiunte al sistema, ma sono presenti dall'inizio, e vengono utilizzate per la gestione delle fluttuazioni short-term.

1.2 Oggetto di studio

L'oggetto di questo studio è un sistema multi-server dotato di un meccanismo di routing dinamico basato sullo stato del sistema e della sua congestione, e che implementa diverse tipologie di risorse. In particolare si hanno:

- **Due tipi di risorse**: le risorse primarie per l'evasione delle richieste sono denominate "**Web Server**", mentre la risorsa secondaria per la gestione delle congestioni è denominata "**Spike Server**".
- **Routing dinamico basato sullo stato**: quando il sistema determina che i Web Server si trovano in una condizione di congestione dovuta ad un improvviso picco di carico, indirizza le richieste in entrata verso la risorsa secondaria, ovverosia lo Spike Server. Questo permette di alleviare la congestione nei WS che hanno tempo di evadere parte delle loro richieste, ed è una soluzione per i problemi introdotti dalle fluttuazioni short-term.

2 Obiettivi

L'obiettivo di questo studio è il **"right-sizing problem"**, cioè identificare il minor numero di risorse che devono essere allocate per ottenere i valori di performance desiderati. Ciò, oltre a non essere un problema banale, rappresenta un'approssimazione del costo dell'infrastruttura in un eventuale contesto cloud.

Più precisamente, dati una caratterizzazione del carico, gli indicatori di QoS da rispettare sono calcolati in funzione del 95-esimo percentile del tempo di risposta delle richieste.
Si vogliono quindi determinare i parametri che permettano di utilizzare il minor numero di risorse pur rispettando gli indicatori di QoS e valutare il risparmio di risorse allocate rispetto all'approccio con "overprovisioning".

Ponendo il focus dello studio sul risparmio delle risorse e sul rispetto dei QoS, si vogliono individuare:

- Il numero di Web Server da allocare.
- Se lo Spike Server porta un impatto positivo.
- La politica di routing dinamico per la gestione delle congestioni.

3 Modello concettuale

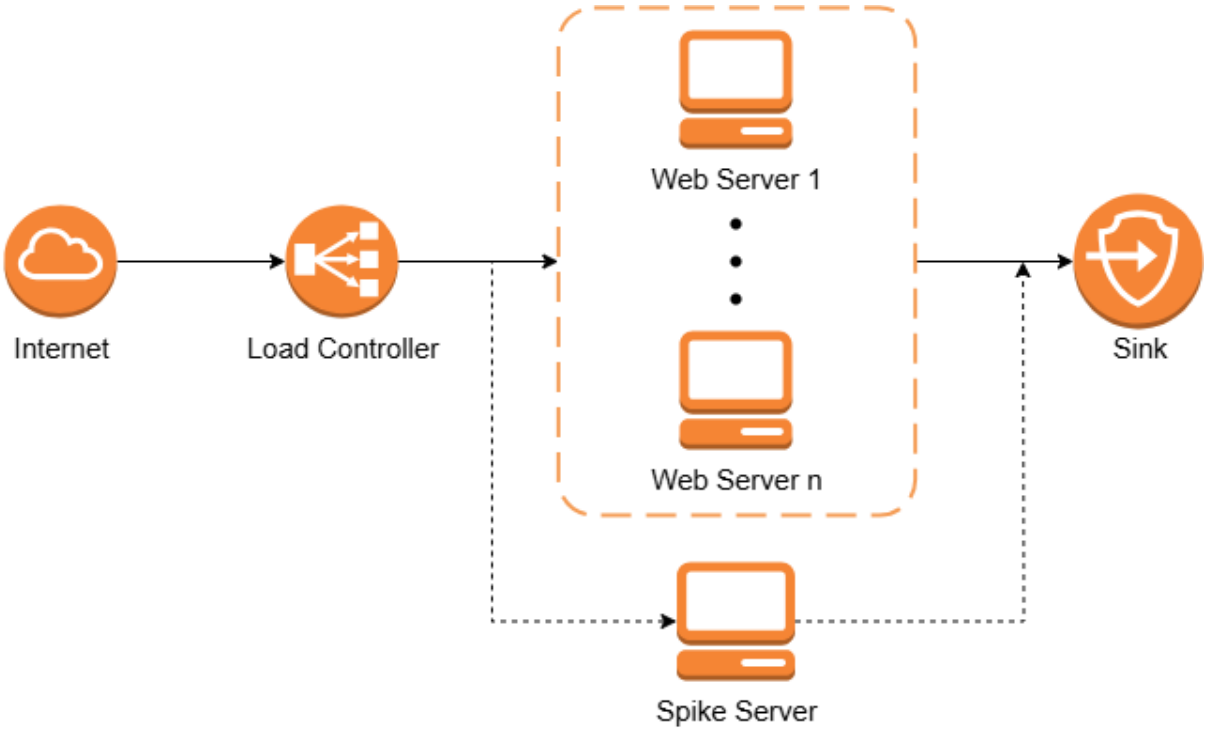

Figura 1: Modello concettuale del sistema

3.1 Lo stato del sistema

Lo stato del sistema è composto dallo stato dei suoi componenti, i quali vengono descritti di seguito.

3.1.1 Servers

M Web Server (WS) totali a disposizione: queste sono le risorse primarie per l'evasione delle richieste in entrata.
1 Spike Server (SS) a disposizione: questa è la risorsa secondaria per l'evasione delle richieste in entrata in caso di congestione delle risorse primarie.

Lo stato di ogni Server è caratterizzato da:

- Una capacità di calcolo, espressa in $\frac{\text{operazioni}}{s}$.
- Un insieme di richieste che il server sta processando.

Sia i Web Servers sia lo Spike Server modellano il comportamento di processori che dividono la capacità computazionale equamente tra tutte le richieste in corso di evasione e seguono quindi una policy di Processor Sharing.

3.1.2 Load controller

1 Load Controller (LC). Questo componente monitora lo stato delle risorse e prende le decisioni di routing.

- Monitora il numero di richieste pendenti e determina se i Web Server si trovano in una condizione normale o in una condizione di congestione, e decide se instradare verso i Web Server o lo Spike Server.
- Se i Web Server non sono stato di congestione, allora divide le richieste tra i Web Server secondo una policy astratta.
- Il tempo di esecuzione del LC è trascurabile e non introduce code. Quindi in ogni istante una qualsiasi richiesta è assegnata ad uno dei Web Server o allo Spike Server.

3.1.3 Job del sistema

Le richieste al sistema, identificabili come i "job", sono generate da una sorgente. Dal momento che lo scheduling dei server è Processor Sharing, è essenziale mantenere uno stato di avanzamento per ciascun job che determina il tempo rimanente di esecuzione in ciascun istante.

3.2 Eventi del sistema

Ad ogni istante, gli **eventi** possibili nel sistema sono:

- Arrivo di un job al sistema: il LC determina in base alle stato delle risorse verso quale di esse indirizzare la richiesta.
- Completamento di un job: il job lascia il sistema.

4 Modello delle specifiche

4.1 Distribuzione inter-arrivi e richieste

I tempi di interarrivo al sistema sono modellati come un'iperesponenziale di media 0.25 s e coefficiente di variazione 4.
Le richieste dei job sono modellate come un'iperesponenziale di media 1000 operazioni e coefficiente di variazione 4.

La scelta dell'iperesponenziale, oltre per coerenza con lo studio iniziale del libro, è stata presa per via della sua caratteristica heavy-tail, grazie alla quale il sistema viene stimolato dalle fluttuazioni short-term che si vogliono studiare. Il motivo della scelta, rispetto al solito tempo di interarrivo/domanda esponenziale, è che si vuole sottoporre il sistema ad una maggiore variabilità di carico e vedere come si comporta durante intervalli di congestione.

4.1.1 Fluttuazioni long-term

Come accennato poco prima, l'heavy-tail della distribuzione iperesponenziale modella fluttuazioni short-term. Per simulare anche fluttuazioni long-term invece è stato definito il seguente scenario, dove periodicamente ogni 500 s, si susseguono:

- Un intervallo di 400 s, durante il quale il tasso di arrivi è "basso", con media λ_{slow}.
- Un intervallo di 100 s, durante il quale il tasso di arrivi è "alto", con media λ_{fast}.

I valori di λ_{slow} e λ_{fast} sono stati scelti per rispettare la seguente condizione: la media del tasso di arrivi al secondo durante tutto il periodo di 500 s deve essere uguale a $\lambda = 4.0\ \frac{\text{arrivi}}{s}$.
In particolare è stato scelto $\lambda_{\text{fast}} = 8.0\ \frac{\text{arrivi}}{s}$ e di conseguenza:

$$\frac{400}{500} \cdot \lambda_{\text{slow}} + \frac{100}{500} \cdot \underbrace{8.0}_{\lambda_{\text{fast}}} = 4.0 \Rightarrow \lambda_{\text{slow}} = 3.0\ \frac{\text{arrivi}}{s}$$

4.2 Modellazione dei server

I singoli server sono modellati come dei centri $G/H_2/1/PS$: infatti, nonostante gli arrivi al sistema siano iperesponenziali, la presenza del routing dinamico tra Web Server e Spike Server cambia la distribuzione degli arrivi ai singoli server, che non si può più dire sia iperesponenziale.

4.2.1 Parametri dei server

Sia Web Server che Spike Server sono stati implementati con una capacità pari a $C = 1000\ \frac{operazioni}{s}$: questo comporta che la distribuzione del tempo di servizio è iperesponenziale con media pari a $E[S] = \frac{E[Z]}{C} = 1\ s$ e coefficiente di variazione pari a 4.

4.2.2 Processor Sharing

Il processor sharing è una politica di servizio in cui si assegna una risorsa ad un processo per un certo quanto di tempo.

Lo stato di un server che implementa PS con tempi di servizio generici può essere descritto tramite:

- Capacità del server C, espressa in $\frac{\text{operazioni}}{s}$.
- Lista con i job in corso di servizio $jobs = \{job_1, job_2, \dots\}$.
  - $|jobs|$ indica il numero di job attualmente in corso di servizio.
  - Il valore di job_i indica la richiesta rimanente del job i-esimo, espressa in operazioni.

È possibile determinare come lo stato evolve fino al successivo arrivo o completamento, assumendo di conoscere gli istanti di arrivo al server, con il seguente metodo:

1. Calcolare l'intervallo di tempo richiesto affinché il primo job del server completi.
   In PS è possibile approssimare l'esecuzione come se la capacità complessiva del server venisse equamente spartita tra tutti i job in servizio: $\frac{C}{|jobs|}$.
   Quindi il primo job a completare sarà quello con la richiesta rimanente minore e completerà dopo un intervallo di lunghezza:
   $$c_{\text{next}} = \min_i \frac{job_i}{\frac{C}{|jobs|}}$$
2. Ottenere l'intervallo di tempo richiesto affinché il prossimo job arrivi al server, indicato con a_{next}.
3. Calcolare $advance = \min(c_{\text{next}}, a_{\text{next}})$.
   È necessario procedere per step, poiché al cambiamento di $|jobs|$ cambia la capacità nominale assegnata ad ogni job.
4. Aggiornare le richieste rimanenti dei job che erano attivi, che diminuiscono della capacità nominale moltiplicata per il tempo in cui questa viene applicata:
   $$\forall\ i:\ job_i = job_i - \frac{C}{|jobs|} \cdot advance$$
   Al più un job potrà avere richiesta rimanente pari a 0 s, e questo può accadere se e solo se $advance = c_{next}$.

4.3 Modellazione del Load Controller

4.3.1 Scheduler intra-server

Gli arrivi al sistema sono gestiti utilizzando la politica di scheduling Least Used, la quale sceglie come server a cui assegnare la richiesta quello con il minor numero di job assegnati.

4.3.2 Routing dinamico: l'indicatore SI

Soluzione 1
Per la decisione del routing dinamico si considera una metrica Spike Indicator (SI) che ha una soglia $SI^{\max}$. Con SI indichiamo il numero di richieste concorrenti nel sistema. Dato un sistema dotato di un certo numero m di Web Server, nel momento in cui diventa vera la condizione

$$SI > m \cdot SI^{\max}$$

ovvero quando nel sistema coesistono almeno $SI^{\max}$ richieste, i nuovi job in arrivo verranno inoltrati verso lo Spike Server finché SI non scenderà nuovamente al di sotto di $SI^{\max}$.

Soluzione 2
Lo studio originale definiva, al momento del superamento della soglia $SI^{\max}$, un'assegnazione totale delle nuove richieste verso lo Spike Server: in presenza di lunghe o intense fluttuazioni però lo Spike Server diventerebbe un collo di bottiglia, poiché verrebbe congestionato. Perciò è stata considerata un'altra soluzione: in caso di congestione dello Spike Server, si decide di trattare tutti i server equamente. In particolare quindi discriminiamo 3 situazioni:

- **Bassa intensità di carico**: $SI < m \cdot SI^{\max}$, i Web Server gestiscono le richieste in entrata.
- **Media intensità di carico**: $m \cdot SI^{\max} \leq SI < (m+1) \cdot SI^{\max}$, mentre i Web Server si decongestionano lo Spike Server gestisce le nuove richieste in entrata.
- **Alta intensità di carico**: $SI > (m+1) \cdot SI^{\max}$, quindi sia Web che Spike Server stanno eseguendo in media almeno $SI^{\max}$ richieste, quindi per non congestionare lo Spike Server le nuove richieste verranno distribuite equamente tra tutti i server.

4.4 SLO

Lo SLO è stato definito sul tempo di risposta dei job e per rispettarlo il sistema deve fare in modo che il 95% dei job abbia un tempo di risposta minore della soglia SLO_{95}.

L'attuale valore di SLO_{95} è una **euristiche** definita utilizzando il seguente approccio:

1. Data la distribuzione del tempo di servizio, si calcola il 95-esimo percentile del tempo di servizio. Questo è un lower bound per la soglia, in quanto approssima i percentili sul tempo di risposta quando il sistema è ha utilizzazione bassissima e i job non incontrano ritardi dovuti agli altri job. In particolare, per l'esponenziale con media $E[S] = 1$, il valori calcolato è $S_{95} = 3.714886\ s$. Questi sono stati calcolati analiticamente a partire dalla densità dell'iperesponenziale.

2. Utilizzando la formula di Kingman si approssima il tempo in coda che un job sperimenterebbe se il sistema fosse $G/G/1$.
   La formula richiede un'utilizzazione del sistema, ed è stata scelta $\rho = 0.6$, che rappresenta un sistema con utilizzazione adeguata.

$$\begin{aligned} E[W_q] &\approx \frac{\rho}{1-\rho} \cdot \frac{c_a^2 + c_s^2}{2} \cdot E[S] \\ &\Downarrow \rho = 0.6, c_a^2 = c_s^2 = 4, E[S] = 1 \\ &\approx \frac{0.6}{1-0.4} \cdot \frac{4+4}{2} \cdot 1 = 6\ s \end{aligned}$$

   Abbiamo utilizzato questa formula in quanto versione più generale della KP, la quale nel nostro contesto non è stata utilizzabile per via degli arrivi non Markoviani.

3. Si fissa SLO_{95} come la somma del percentile del tempo di servizio e del tempo medio in coda:

$$\text{SLO}_{95} = S_{95} + E[W_q] = 3.714886 + 6 = 9.714886\ s$$

5 Modello computazionale

La simulazione, di tipo **next-event**, è stata implementata utilizzando Java come linguaggio di programmazione.

5.1 Premessa

Per semplicità, la configurazione del sistema base è impostata con richiesta media dei job pari ad 1 operazione e capacità dei Server pari ad 1 $\frac{\text{operazioni}}{s}$ in modo che la distribuzione delle richieste dei job corrisponda alla distribuzione dei tempi di servizio.

5.2 La gestione degli eventi

Gli eventi nel sistema sono stati modellati attraverso la seguente classe astratta:

```java
public abstract class Event {
    @Getter private double timestamp;
    @Getter private EventType eventType;

    protected Event(double timestamp, EventType eventType) {
        this.timestamp = timestamp;
        this.eventType = eventType;
    }

    public abstract void process(SystemState s, EventVisitor visitor) throws IllegalLifeException;
}
```

Listing 1: Classe astratta che modella gli eventi nel sistema

L'attributo `eventType` rappresenta il tipo di evento che si vuole processare. Invece l'attributo `timestamp` rappresenta il tempo in cui l'evento si è verificato. I tipi di evento possibili sono:

- Arrivo
- Completamento

5.2.1 Evento di arrivo

```java
public void visit(SystemState s, ArrivalEvent event) {
    IServerInfrastructure servers = s.getServers();

    /* Get the current clock and the one of this arrival */
    double startTs = s.getCurrent();
    double endTs = event.getTimestamp();

    /* Advance job execution */
    servers.computeJobsAdvancement(startTs, endTs, false);

    /* Add the next job to the list */
    double nextServiceLife = s.getServicesVA().gen();
    Job newJob = new Job(endTs, nextServiceLife);
    servers.assignJob(newJob);

    /* Generate next completion */
    double nextCompletionTs = servers.computeNextCompletionTs(endTs);
    Event nextCompletion = new CompletionEvent(nextCompletionTs);
    s.addEvent(nextCompletion);

    /* Generate next arrival if simulation is not finished*/
    if (endTs < STOP) {
        double nextArrivalTs = endTs + s.getArrivalVA().gen();
        Event nextArrival = new ArrivalEvent(nextArrivalTs);
        s.addEvent(nextArrival);
    } else {
        s.addEvent(new ArrivalEvent(INFINITY));
    }

    /* Update the current system clock */
    s.setCurrent(endTs);
}
```

Listing 2: Codice che implementa evento di arrivo.

Quando si verifica un evento di arrivo:

1. Si recupera il timestamp corrente e quello in cui si è verificato l'evento.
2. Viene calcolato l'avanzamento dei job specificando che non si tratta di un completamento.
3. Viene assegnato il job arrivato ad un server
4. Si ricalcola il tempo del prossimo completamento, poiché per il server a cui è stato assegnato esso varia a causa del processor sharing.
5. Se la simulazione non è terminata, si rigenera il prossimo arrivo.
6. Si aggiorna il clock del simulazione.

5.2.2 Evento di completamento

```
public void visit(SystemState s, CompletionEvent event)  {
    IServerInfrastructure servers = s.getServers();

    /* Get the current clock and the one of this arrival */
    double startTs = s.getCurrent();
    double endTs = event.getTimestamp();

    /* Advance job execution */
    servers.computeJobsAdvancement(startTs, endTs, true);

    /* Generate next completion */
    double nextCompletionTs = servers.activeJobExists() ?
        servers.computeNextCompletionTs(endTs) :
        INFINITY;
    Event nextCompletion = new CompletionEvent(nextCompletionTs);
    s.addEvent(nextCompletion);

    /* Update the current system clock */
    s.setCurrent(endTs);
}
```

Listing 3: Codice che implementa evento di completamento.

Quando si verifica un evento di completamento:

1. Si recupera il timestamp corrente e quello in cui si è verificato l'evento.
2. Viene calcolato l'avanzamento dei job specificando che si tratta di un completamento.
3. Si ricalcola il tempo del prossimo completamento, poiché per il server che ha completato esso varia a causa del processor sharing.
4. Si aggiorna il clock del simulazione.

5.3 Stato del sistema: la classe SystemState

Lo stato del sistema è mantenuto nella classe `SystemState`, nella quale troviamo:

- `current`, un `double` che rappresenta clock della simulazione.
- `calendar`, una struttura dati per implementare la next-event simulation.
- `servers`, una classe che rappresenta l'infrastruttura dei server.

5.3.1 Gestione della coda degli eventi: la classe EventCalendar

Come richiesto dalla next-event simulation, viene mantenuto il riferimento solo al prossimo evento per ogni tipo. Nel codice questo è fatto dalla classe `Calendar`, in particolare dal metodo `nextEvent()` che ritorna l'evento con il timestamp minore, come indicato di seguito:

```java
public class EventCalendar {
    /*
    * Map of the next event for each event type
    */
    private final Map<EventType, Event> events = new HashMap<>();

    /**
     * Adds event in the map
     * @param e The event to add
     */
    public void addEvent(Event e) {
        events.put(e.getEventType(), e);
    }

    /**
     * Get event with minimum timestamp
     */
    public Event nextEvent() {
        return events.values()
                .stream()
                .min(Comparator.comparing(Event::getTimestamp))
                .orElse(null);
    }
}
```

Listing 4: Classe Event Calendar

5.3.2 Comportamento dell'infrastruttura del sistema: le classi ServerInfrastructure

Per determinare come lo stato debba evolvere è stata definita un'interfaccia che astrae il comportamento del sistema, la `IServerInfrastructure`, la quale si può scegliere se implementarla senza lo Spike Server (utilizzando la classe `BaseServerInfrastructure`) o aggiungerci le funzionalità dello SpikeServer (tramite pattern Decorator con la classe `SpikedServerDecorator`) in modo da poter studiare i due sistemi separatamente.
Ogni ServerInfrastructure contiene gli indicatori per il routing dinamico e la lista di `Server` a cui assegnare i job.

5.3.3 Astrazione dei server: le classi Server

Si usa la classe `AbstractServer` per definire tutte le componenti in comune tra i server, ovvero la propria capacità, la lista di job assegnati ad ognuno di essi e l'implementazione dell'avanzamento dei job con la politica processor sharing. La classe astratta viene poi estesa dalle due tipologie di Server possibili, `WebServer` e `SpikeServer`, ognuna con i suoi comportamenti più specifici.

5.3.4 Astrazione dei job: le classi JobList e Job

Ogni server ha una classe `JobList` che rappresenta l'insieme dei job assegnati ad esso, e questa classe contiene a sua volta una lista di `Job` da gestire. Ciascun elemento di questa lista è costituito da due attributi:

- `arrivalTime`: istante in cui il job è arrivato nel sistema.
- `remainingLife`: tempo d'esecuzione rimanente per il job.

La life di un job verrà decrementata, in funzione del tempo trascorso, durante il calcolo dell'avanzamento dei job.

5.4 Avanzamento dei job

Lo stesso avanzamento dei job viene utilizzato sia per gli eventi di arrivo che di completamento. Se si tratta di un completamento, viene aggiunta la rimozione a monte dell'avanzamento dei job: questa operazione viene effettuata in anticipo per risolvere un problema di instabilità nella precisione numerica dei calcoli, il quale causava la presenza di job con una life negativa. Nel caso non si tratti di un completamento, ma di un arrivo, viene semplicemente calcolato l'avanzamento dei job su tutti i server in base al tempo trascorso.

```
public double computeJobsAdvancement(double startTs, double endTs, boolean isCompletion) {
    Integer completionServerIndex = null;
    Double completedJobResponseTime = null;
    Job completedJob;

    /* If there is a completion, before updating the jobs life
       get the completing job to take its stats and remove it*/
    if (isCompletion) {
        completionServerIndex = getCompletingServerIndex();
        WebServer minServer = webServers.get(completionServerIndex);

        completedJob = minServer.getMinRemainingLifeJob();
        completedJobResponseTime = endTs - completedJob.getArrivalTime();

        minServer.removeJob(completedJob);
        ...
    }

    /* Compute the advancement of each job in each web Server */
    for (int currIndex = 0; currIndex < servers.size(); currIndex++) {
        AbstractServer server = webServers.get(currIndex);
        server.computeJobsAdvancement(
                startTs, endTs,
                Objects.equals(currIndex, completionServerIndex) ? completedJobResponseTime : null);
    }
    ...
}
```

Listing 5: Implementazione avanzamento dei job nell'infrastruttura

Ogni server, indipendentemente dagli altri, calcola l'avanzamento dei propri job secondo **PS** come indicato precedentemente e aggiorna le proprie statistiche.

```
public void computeJobsAdvancement(double startTs, double endTs, Double completedJobResponseTime) {
    /*  At this point the job has already been removed, so if this is the server
     *  where completion happened, it must be taken into consideration */
    int completedJob = completedJobResponseTime == null ? 0 : 1;
    int jobAdvanced = jobs.size() + completedJob;
    ...

    /* Compute the advancement of each job */
    double quantum = (this.capacity / jobAdvanced) * (endTs - startTs);
    try{
        for (Job job : this.jobs.getJobs()) {
            job.decreaseRemainingLife(quantum);
        }
    } catch (IllegalLifeException e) {
        System.out.printf("endTs: %f", endTs);
        throw e;
    }
}
```

Listing 6: Implementazione avanzamento dei job nei server

```java
/**
 * @param jobNum the number of jobs in the server from startTs to endTs
 * @param completedJobResponseTime null if no completion, otherwise the response time of the completed job an endTs
 */
public void updateServerStats(
    double startTs, double endTs,
    double jobNum, Double completedJobResponseTime) {

    /* Check if there has been a completion */
    boolean isCompletion = completedJobResponseTime != null;

    /* Update statistics */
    if(jobNum > 0) {
        this.nodeSum    += (endTs - startTs) * jobNum;
        this.serviceSum += (endTs - startTs);
    }

    if (isCompletion){
        this.completedJobs++;

        /* Update SLO indicator */
        if (completedJobResponseTime <= RESPONSE_TIME_95PERC_SLO)
            this.jobRespecting95percSLO++;

        /* Update mean response time */
        currMeanResponseTime += (completedJobResponseTime - currMeanResponseTime) / completedJobs;
    }
}
```

Listing 7: Implementazione aggiornamento statistiche di un server

5.5 Le variabili aleatorie

Alla base dell'implementazione delle variabili aleatorie troviamo la classe `Rngs` progettata per la generazione di numeri pseudo-casuali multi-stream, in maniera più robusta rispetto alle librerie offerte da `Java`. Tale classe utilizza un **generatore di Lehmer** che con il multi-stream permette di generare dei numeri pseudo-random indipendenti.
Nel modello base questa libreria è stata utilizzata per modellare i tempi di servizio e di arrivo attraverso delle variabili aleatorie iper-esponenziali.

5.5.1 Gestione dei seed

Lo stato di ciascuno stream è impostato attraverso la chiamata

```
rngs.plantSeed(int x)
```

dove `x` è il valore iniziale del seed. In particolare, attraverso questa funzione è possibile andare a settare lo stato di tutti gli stream "piantando" una sequenza di stati (seed).

5.5.2 La variabile iper-esponenziale

```
public class DistributionFactory {

    public static Distribution createArrivalDistribution(Rngs rngs) {
        int stream0 = 0;
        int stream1 = 1;
        int stream2 = 2;
        return switch (ARRIVALS_DISTR) {
            case "exp" -> new Exponential(rngs, ARRIVALS_MU, stream0);
            case "h2"  -> new CHyperExponential(rngs, ARRIVALS_CV, ARRIVALS_MU, stream0, stream1, stream2);
        };
    }

    public static Distribution createServiceDistribution(Rngs rngs) {
        int stream0 = 3;
        int stream1 = 4;
        int stream2 = 5;
        return switch (SERVICES_DISTR) {
            case "exp" -> new Exponential(rngs, SERVICES_Z, stream0);
            case "h2"  -> new CHyperExponential(rngs, SERVICES_CV, SERVICES_Z, stream0, stream1, stream2);
        };
    }
}
```

Listing 8: Creazione distribuzioni arrivi e servizi con rng multi-stream

Per implementare la variabile aleatoria iper-esponenziale è stato necessario utilizzare:

- Due variabili aleatorie esponenziali con parametri diversi.
- Una variabile aleatoria di Bernoulli: utilizzata per decidere quale ramo di esponenziale scegliere in ciascun campionamento.

Tutte e 3 queste variabili utilizzano degli stream diversi in modo da usare generatori indipendenti: come da codice 8, gli stream 0, 1 e 2 sono usati per gli inter-arrivi, mentre 3, 4 e 5 per le richieste dei job.

L'implementazione della variabile iperesponenziale è funzione del coefficiente di variazione C^2 e della media della variabile iper-esponenziale $\frac{1}{\mu_X}$. In questo secondo caso per generare i numeri abbiamo sempre bisogno di ricostruire le variabili esponenziali e la bernouliana. Ovvero dobbiamo ricavare i parametri delle tre distribuzioni. Per fare questo sono state utilizzate le seguenti formule:

$$\mu_1 = \mu_X \cdot 2p$$
$$\mu_2 = \mu_X \cdot 2(1-p)$$

Per verificare la validità di queste formule basta sostituire μ_1 e μ_2 in:

$$\begin{aligned} E[X] &= \frac{1}{\mu_1} \cdot p_1 + \frac{1}{\mu_2}(1-p) \\ &= \frac{1}{\mu_X \cdot 2p} p + \frac{1}{\mu_X \cdot 2(1-p)}(1-p) \\ &= \frac{1}{2\mu_X} + \frac{1}{2\mu_X} = \frac{1}{\mu_X} \end{aligned}$$

Date queste due relazioni possiamo ricavare p a partire da:

$$
\begin{aligned}
C^2 &= \frac{1}{2p(1-p)} - 1 \\
p(1-p) &= \frac{1}{2(C^2+1)} \\
&\Downarrow p(1-p) = \frac{1}{4}\left[1-(2p-1)^2\right] \\
1-(2p-1)^2 &= \frac{2}{C^2+1} \\
2p-1 &= \sqrt{\frac{C^2-1}{C^2+1}} \\
p &= \frac{1}{2}\left(1+\sqrt{\frac{C^2+1}{C^2-1}}\right)
\end{aligned}
$$

In questo modo è possibile ripetere gli esperimenti del libro dato che per l'iperesponenziale vengono dati come parametri il coefficiente di variazione e la media.

5.6 Le configurazioni del sistema

Attraverso un file di configurazione `config.properties` è possibile scegliere i parametri del sistema, tra i quali ci sono: i parametri delle variabili aleatorie, i seed, il numero di server nel sistema, ecc..

Attraverso la classe `Config` è stato implementato un gestore centralizzato delle configurazioni del sistema che ci permette di:

1. Caricare i parametri dal file di configurazione oppure da un `RunConfigurator`, il quale permette di sovrascrivere il contenuto del file e fare esperimenti con diverse configurazioni.
2. Salvare i valori di tutti parametri di configurazione su CSV, per avere ripetibilità delle run.
3. Usando la `ExperimentFactory` genera i parametri d'esecuzione per gli esperimenti.

5.7 Raccolta dei dati e logging

Esistono due tipi di dati:

- Dati intra-run. Questi vengono raccolti in corrispondenza di ogni evento, e si suddividono a loro volta in:
  - Dati intra-run generici. Questi vengono utilizzati per operazioni di debugging e validazione.
  - Dati intra-run aggregati. Questi vengono utilizzati per lo studio del transitorio.
- Dati inter-run. Questi vengono raccolti al termine della run e definiscono statistiche globali della run

La raccolta dei dati viene effettuata utilizzando la classe `DataTimeTable` che permette di utilizzare un `double` come chiave e di aggiungere un numero arbitrario di parametri con il relativo valore.
Per i dati intra-run, la chiave corrisponde al timestamp dell'evento di cui si vuole raccogliere i dati.
Per i dati inter-run c'è una chiave speciale univoca per l'esecuzione della run.
È possibile disattivare la raccolta di dati intra-run per ridurre l'impatto sullo storage in caso di un numero elevato di simulazioni, con il parametro di configurazione `log.intra_run`.

5.7.1 Esempi

Un esempio di raccolta dati intra-run è il seguente, dove viene raccolto il tempo di risposta del job che ha appena completato in un evento di completamento:

```java
public double computeJobsAdvancement(double startTs, double endTs, boolean isCompletion) {
    ...
    if (isCompletion) {
        INTRA_RUN_DATA.addField(endTs, R_0, completedJobResponseTime);
    }
    ...
}
```

Un esempio importante di raccolta di dati inter-run è il seguente, dove vengono raccolti i seed di ogni stream ad inizio run per riproducibilità[1]:

```java
/* Log to CSV the initial seed for each stream for replayability */
for (int stream = 0; stream < TOTAL_STREAMS; stream++) {
    R.selectStream(stream);
    INTER_RUN_DATA.addFieldWithSuffix(
        INTER_RUN_KEY, STREAM_SEED,     // key
        String.valueOf(stream),         // field
        R.getSeed()                     // value
    );
}
```

5.7.2 Dati raccolti

Alcuni tra i dati intra-run che vengono raccolti sono i seguenti:

- `TIMESTAMP`: Timestamp dell'evento.
- `EVENT_TYPE_JOB`: Tipo dell'evento.
- `JOBS_IN_SYSTEM`: Numero di job nel sistema.
- `JOBS_IN_SERVER_i`: Numero di job nel server i-esimo.
- `R_0`: tempo di risposta del job (se completamento).
- `COMPLETING_SERVER_INDEX`: indice del server che ha completato il job (se completamento).
- `JOB_SIZE`: Size generata del job arrivato (se arrivo).
- `NEXT_INTERARRIVAL_TIME`: Tempo di interarrivo del prossimo job (se arrivo).

  ...

Alcuni tra i dati inter-run che vengono raccolti sono i seguenti:

- `STREAM_SEED_i`: Seed dello stream i-esimo all'inzizio della simulazione.
- `nome del parametro`: Valore del parametro di configurazione indicato dal nome per riproducibilità.[2]
- `TOTAL_JOBS_COMPLETED`: Numero totale dei job completati durante l'esecuzione.
- `TOTAL_SPIKE_JOBS_COMPLETED`: Numero totale dei job completati durante l'esecuzione esclusivamente dallo Spike Server.
- `TOTAL_ALLOCATED_CAPACITY`: Totale della capacità allocata del sistema durante l'esecuzione, espressa in $\frac{\text{operazioni}}{s} \cdot s$.
- `SLO_95PERC_VIOLATIONS_PERCENTAGE`: Percentuale delle violazioni dello SLO su 95-esimo percentile rispetto al numero di totale di jobs completati.

  ...

5.7.3 Logging su CSV

I dati, una volta raccolti, possono essere salvati su file in formato `.csv` tramite la classe `DataCSVWriter`. Questa classe permette di specificare quali campi salvare e con che nome salvare il file, in questo modo è possibile produrre diversi file con diverse viste sui dati.

[1] MANET simulation studies: the incredibles
[2] MANET simulation studies: the incredibles

```
DataHeaders jobsHeaders = new DataHeaders();
jobsHeaders.add(
        TIMESTAMP,
        EVENT_TYPE_JOB,
        PER_JOB_RESPONSE_TIME,
        COMPLETING_SERVER_INDEX, JOBS_IN_SYSTEM,
        EVENT_TYPE_SCALING, SCALING_INDICATOR,
        JOB_SIZE, NEXT_INTERARRIVAL_TIME
);

flushList(
    INTRA_RUN_DATA, "jobs_data.csv",
    jobsHeaders.get(), false
);
```

5.8 Esperimenti

5.8.1 Replicazione

Per fare un'analisi del transitorio è stata aggiunta la possibilità di fare più repliche di una run con una specifica configurazione di parametri. Per la correttezza della simulazione, ogni run deve però avere un seed iniziale tale da non avere overlap, nel processo di generazione dei numeri random, con le run precedenti: per realizzare questo si mantiene la stessa istanza del generatore, inizializzata soltanto alla prima run. In questo modo ogni replica consuma un tratto contiguo e disgiunto della stessa sequenza pseudocasuale, evitando qualsiasi overlap nei numeri utilizzati dalle diverse repliche, e al tempo stesso garantendo anche la riproducibilità della simulazione.

```
private static final Rngs R = new Rngs();

public static void main(String[] args) {
    List<RunConfiguration> configurations = Config.createConfigurations();
    for (RunConfiguration c : configurations) {
        setup(c);
        for (int i = 0; i < REPEAT_CONFIGURATION; i++) {
            run(c, i);
            log(c, i);
        }
    }
}
...
private static void run(RunConfiguration c, int repetition) {
    ...
    /* Plant the seeds only if the first running the new configuration */
    if (repetition == 0) {
        R.plantSeeds(SEED);
    }

    EventVisitor visitor = new EventProcessor();

    Distribution arrivalVA = DistributionFactory.createArrivalDistribution(R);
    Distribution servicesVA = DistributionFactory.createServiceDistribution(R);
    ...
```

Listing 9: Replicazione

Le statistiche transienti sono sviluppate all'interno della classe `TransientStats`, la quale calcola ad ogni evento che accade al tempo t le statistiche:

- Tempo medio di risposta.
- Popolazione media.
- Utilizzazione media.
- Capacità media allocata al secondo.

Queste statistiche sono calcolate per ogni Server e anche per il sistema in generale, in modo da poter vedere le performance sia dell'insieme che dei singoli componenti.

5.8.2 Batch means e intervalli di confidenza

Per fare un'analisi della stazionarietà è stata sviluppata la classe `StationaryStats`, nella quale è inglobato il meccanismo dei batch means e degli intervalli di confidenza. Negli esperimenti dello stazio-

nario questa classe andrà a catturare periodicamente (dove il periodo è definito nella variabile globale STATS_BATCH_SIZE) la media delle metriche da voler riportare per ogni batch, tra cui

- Tempo di risposta.
- Popolazione.
- Utilizzazione.
- Capacità allocata al secondo.
- Percentuale dei completamenti che violano lo SLO al 95 percento.

Si riporta di seguito come viene calcolata ciascuna metrica, per semplicità verrà considerato solo il tempo di risposta, le altre statistiche sopra descritte sono state calcolate analogamente.
La classe mantiene degli attributi per il calcolo delle statistiche, in particolare la somma di tutte le medie dei vari batch, il batch corrente, e le variabili che terminano con X e V sono usate per il calcolo della media e varianza online tramite **algoritmo di Welford**.

```
public class StationaryStats {
    private int counter;               /* For tracking when to trigger update*/
    private int currBatch;             /* For tracking the last processed batch */
    private final Integer serverIndex; /* Null if System stats, not null if server stats */

    /* Metrics sums */
    private double sumBatchesResponseTime;

    /* Welford algorithm variables*/
    private double responseTimeX;
    private double responseTimeV;
    ...

    public void printIntervalEstimation() {
        double u, t, w;
        double responseTimeS  = Math.sqrt(this.responseTimeV / this.currBatch);

        double confidence = 1 - STATS_CONFIDENCE_ALPHA;
        Rvms rvms = new Rvms();
        u = 1.0 - 0.5 * STATS_CONFIDENCE_ALPHA;       /* interval parameter  */
        t = rvms.idfStudent(this.currBatch - 1, u);   /* critical value of t */

        w = t * responseTimeS / Math.sqrt(this.currBatch - 1);  /* interval half width */

        System.out.print("The expected value of Response Time is in the interval " +
            responseTimeX + " +/- " + w + " with confidence " + confidence);
    }
}
```

Listing 10: Attributi per il calcolo delle statistiche stazionarie tramite intervalli di confidenza

6 Verifica

Per accertarsi di aver costruito il sistema nel modo giusto si vuole rispondere alle seguenti domande:

1. Gli invarianti sono rispettati?
2. Il clock è strettamente crescente?
3. I risultati ottenuti dal nostro sistema sono coerenti con i risultati analitici?
4. La definizione delle variabili usate nella generazione dei numeri random è corretta?

6.1 Domanda 1

Un invariante nel sistema è ad esempio il numero degli arrivi e il numero di completamenti, che devono essere uguali quando il sistema viene svuotato. Per controllare la consistenza di questo fatto si è eseguita la simulazione con:

- Seed pari a 42.
- Arrivi iperesponenziali con $\mu = 0.15$ e coefficiente di variazione pari a 4
- Domande iperesponenziali con $z = 0.16$ e coefficiente di variazione pari a 4.
- Tempo di simulazione pari a 1000 s.
- La simulazione termina quando il sistema si svuota completamente.

e si può vedere che sia gli arrivi che i completamenti sono esattamente 6790. Da notare che non si è specificata la configurazione dei server dal momento che questo risultato ne è indipendente.

6.2 Domanda 2

Il clock monotono crescente è stato controllato a livello di codice tramite un `assert`.

6.3 Domanda 3

Per poter effettuare un confronto tra i risultati del modello simulativo e quello analitico della teoria è necessario apportare alcune semplificazioni: questo perché la presenza del routing dinamico rende ciascun server un centro $G/H_2/1/PS$, e quindi i risultati teorici affrontati durante il corso non sono utilizzabili. Motivo per cui si semplifica il sistema configurandolo di base nel seguente modo:

- Seed pari a 42.
- Lo Spike Server è spento.
- 1 Web Server di capacità 1 $\frac{\text{operazioni}}{s}$.
- Batch means con 64 batch di size 512.

Vogliamo simulare un centro $M/G/1/PS$ del quale possiamo dire di conoscere, dati λ e μ:

$$\rho = \frac{\lambda}{\mu}$$
$$E(T_S) = \frac{1}{\mu - \lambda}$$
$$E(N_S) = \lambda \cdot E(T_S)$$

Fissando $\mu = 1$ i valori teorici in funzione di λ diventano:

λ	ρ	$\mathbf{E(T_S)}$ (s)	$\mathbf{E(N_S)}$
0.6	0.6	2.5	1.5
0.8	0.8	5	4

I risultati ottenuti sono visibili eseguendo la simulazione dopo aver impostato nel file di `config.properties`

```
experiment=ver_b
```

Tabella 1: Configurazioni esperimenti di verifica

Esperimento	**Distr(λ)**	**Distr(μ)**	λ	μ
ver-b-01	exp	exp	0.6	1
ver-b-02	exp	exp	0.8	1
ver-b-03	exp	h2(cv=4)	0.6	1
ver-b-04	exp	h2(cv=4)	0.8	1

Tabella 2: Risultati con intervalli di confidenza al 95%

Esperimento	ρ	**E(T_S)** (s)	**E(N_S)**
ver-b-01	0.602121 ± 0.008058	2.510876 ± 0.098163	1.5129 ± 0.066007
ver-b-02	0.802826 ± 0.01071	5.034897 ± 0.400671	4.044949 ± 0.345428
ver-b-03	0.594827 ± 0.012792	2.461972 ± 0.151784	1.483423 ± 0.098678
ver-b-04	0.793085 ± 0.016303	4.581886 ± 0.495719	3.680902 ± 0.415406

Come si può vedere i valori ottenuti sono coerenti con quelli teorici.

6.4 Domanda 4

In una simulazione, per asserirne la correttezza, è fondamentale verificare il corretto funzionamento degli RNG utilizzati: da questo punto di vista, la libreria `Rngs` tratta dal libro "*Discrete Event Simulation: A First Course*" di L.M. Leemis e S. Park è una libreria nota e affidabile, quindi rimane da vedere se l'utilizzo di questi strumenti è corretto. A tale scopo è stato ideato un pacchetto di test per ciascuna distribuzione utilizzata.
Tutti i test vanno a buon fine. A titolo di esempio riportiamo, nei codici 11 e 12, i test fatti per la media e la varianza della variabile aleatoria iper-esponenziale.

```
public void testH2() {
    Rngs r = new Rngs();
    r.plantSeeds(SEED);

    // Hyperexponential creation
    CHyperExponential ch2 = new CHyperExponential(r, cv, mean, stream1, stream2, stream3);

    // Width of CI such that P[|X - E[X] > eps] <= alpha
    double eps = Math.sqrt((Math.pow(mean, 2) * cv) / (n * alpha));

    // Sample Mean
    double sampleMean = computeSampleMean(ch2, n);

    // With probability (1 - alpha) this test pass
    assertTrue(message, mean >= sampleMean - eps && mean <= sampleMean + eps);
}
```

Listing 11: Test per la media della distribuzione iperesponenziale

```java
public void testH2Variance() {
    Rngs r = new Rngs();
    r.plantSeeds(SEED);

    // Hyperexponential creation
    CHyperExponential ch2 = new CHyperExponential(r, cv, mean, stream1, stream2, stream3);

    // Width of CI such that P[|X - E[X] > eps] <= alpha
    double eps = Math.sqrt((Math.pow(mean, 2) * cv) / (n * alpha));

    double sampleMean = computeSampleMean(ch2, n);

    // Compute theoretical variance from definition of C^2
    double variance = cv * Math.pow(mean, 2);

    // With probability (1 - alpha) this test pass
    assertTrue(message,
        variance >= cv * Math.pow(sampleMean - eps, 2) &&
        variance <= cv * Math.pow(sampleMean + eps, 2));

    System.out.println(message);
}
```

Listing 12: Test per la varianza della distribuzione iperesponenziale

7 Validazione

Per accertarsi di aver costruito il sistema giusto si vuole rispondere alle domande:

1. Il sistema creato è coerente con i risultati ottenuti dallo studio originale?
2. Lo Spike Server effettivamente aiuta a decongestionare il sistema durante fluttuazoni short-term?
3. Aumentare/diminuire il numero di server migliora/peggiora le performance?

I risultati della validazione sono visibili eseguendo la simulazione dopo aver impostato nel file di `config.properties`

```
experiment=val_b
```

7.1 Domanda 1

Il sistema creato è coerente con i risultati ottenuti dallo studio originale?
Per validare questa condizione configuriamo la simulazione per essere coerente con lo studio originale. Per cui si configura il sistema nel seguente modo:

- Seed pari a 42.
- Lo Spike Server è acceso con $SI^{\max} = 140$.
- Un Web Server di capacità 1 $\frac{\text{operazioni}}{s}$.
- Tempi di interarrivo iperesponenziali di media 0.15 e cv pari a 4.
- Domande delle richieste iperesponenziali di media 0.16 e cv pari a 4.

Il risultato della simulazione è visibile in figura 2: come si può vedere, il comportamento ottenuto è del tutto analogo a quello dello studio originale, riportato in figura 3.

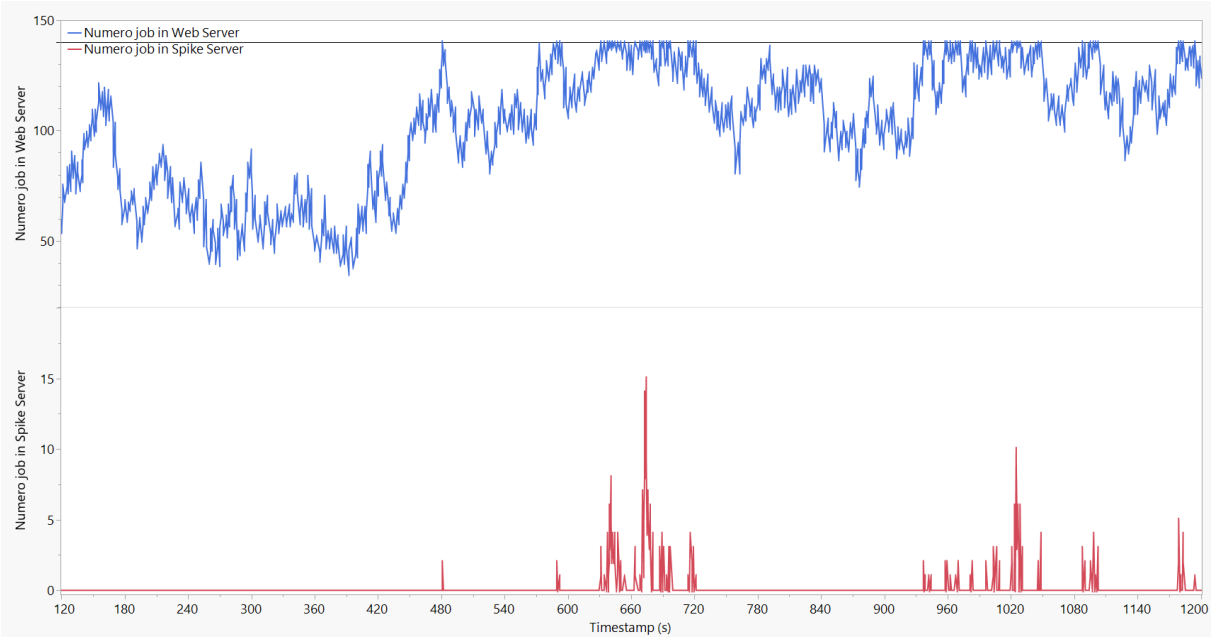

Figura 2: Esperimento val_b_01

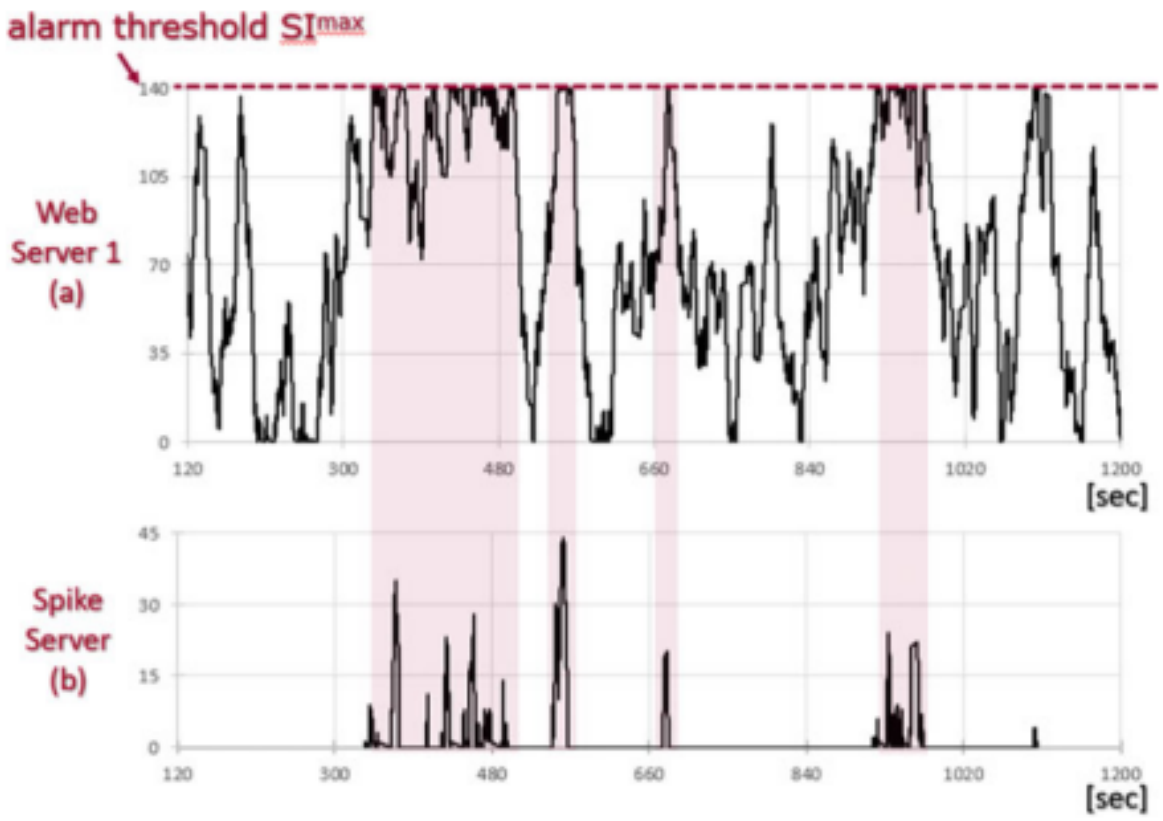

Figura 3: Esperimento originale

7.2 Domanda 2

Lo Spike Server effettivamente aiuta a decongestionare il sistema durante fluttuazoni short-term?
Dal grafico in figura 2 nella pagina precedente possiamo osservare che, in corrispondenza dei picchi nel numero di job nel Web Server, lo Spike Server inizia ad agire e smaltire il carico di lavoro al posto del Web Server, riportandolo ad un numero di job inferiore a 140. Questo comportamento è coerente con quello che ci si aspetta dal sistema implementato.

7.3 Domanda 3

Aumentare/diminuire il numero di server migliora/peggiora le performance?
Per analizzare questo fatto andiamo a considerare la medesima configurazione dell'esperimento in fase di verifica ver_b_01, ma con due server, e osserviamo cosa accade al numero di job nel sistema e al tempo di risposta.

Tabella 3: Esperimenti validazione

Esperimento	**Distr(λ)**	**Distr(μ)**	λ	μ	Num. WS
ver-b-01	exp	exp	0.6	1	1
val-b-02	exp	exp	0.6	1	2

Tabella 4: Intervalli di confidenza al 95%

Esperimento	ρ	**E($\mathbf{T_S}$)** (s)	**E($\mathbf{N_S}$)**
ver-b-01	0.602121 ± 0.008058	2.510876 ± 0.098163	1.5129 ± 0.066007
val-b-02	0.604531 ± 0.005955	1.142166 ± 0.012234	0.689682 ± 0.009279

L'utilizzazione del sistema rimane la stessa, ma il tempo medio di risposta di un job, così come il numero di job presenti nel sistema scende drasticamente, e questo è corretto in confronto alla teoria.

8 Studio del transitorio

In questa fase si mostrano lo studio del transitorio di alcune configurazioni, per assicurare che i parametri utilizzati negli esperimenti presenti nella sezione successiva corrispondano a simulazioni le cui statistiche convergono.

Due metriche importanti che permettono di predire se il sistema convergerà sono le seguenti:

- $E[R]$ Richiesta totale per secondo: espressa in $\frac{\text{operazioni}}{s}$, rappresenta la richiesta totale dei job in unità di tempo.

$$E[R] = E[Z] \cdot \lambda$$

- C Capacità allocata totale per secondo: espressa in $\frac{\text{operazioni}}{s}$, è la somma delle capacità dei server allocati (sia Web Server, sia Spike Server).

$$C = \sum_{\text{Server allocati i}} C_i$$

In generale, dagli studi condotti sembra emergere che le statistiche del sistema convergono se il sistema ha una capacità allocata per secondo maggiore della richiesta per secondo.

$$\text{sistema converge} \iff C > E[R]$$

In tutti gli studi vengono fissati i seguenti parametri:

- Distribuzioni iperesponenziali di arrivi e richieste per job, con i seguenti parametri.

$$\begin{aligned} E[\lambda] &= 4 \ \frac{\text{arrivi}}{s} \\ E[Z] &= 1 \ \text{operazioni} \\ &\Downarrow \\ E[R] &= 4 \ \frac{\text{operazioni}}{s} \end{aligned}$$

$$\begin{aligned} CV_{\text{arrivi}} &= 4.0 \\ CV_{\text{richieste}} &= 4.0 \end{aligned}$$

- Fluttuazioni long-term con periodo di 500 s e i seguenti parametri:

$$\begin{aligned} \lambda_{\text{slow}} &= 3 \ \frac{\text{arrivi}}{s} \ \text{per } 400 \ s \\ \lambda_{\text{fast}} &= 8 \ \frac{\text{arrivi}}{s} \ \text{per } 100 \ s \end{aligned}$$

I risultati di questi studi del transitorio sono visibili eseguendo la simulazione dopo aver impostato nel file di `config.properties`

```
experiment=trans_b
```

8.1 Studio 1

In questo studio si mostra la convergenza delle statistiche di un sistema con:

- La versione `1.0` del routing dinamico.
- 6 Web server, e quindi rispetta la condizione $C > E[R]$.
- Soglia di congestione $SI^{\max} = 5.0$

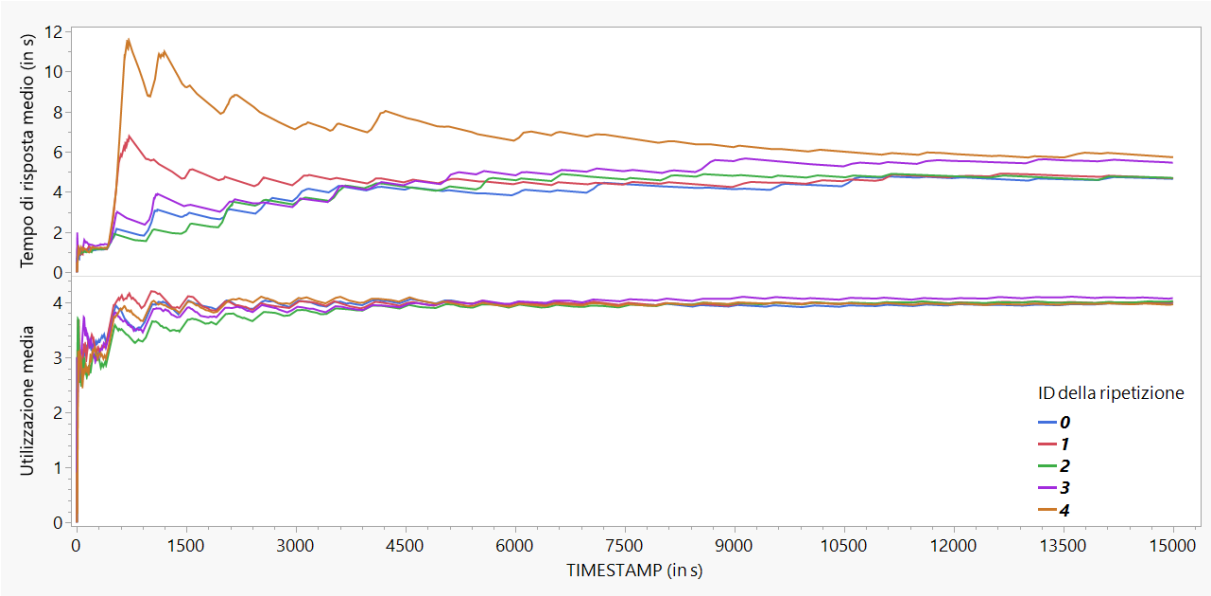

Figura 4: Studio del transitorio della configurazione 1

I risultati in formato tabellare sono stati ottenuti con 64 ripetizioni e gli intervalli di confidenza al 95% calcolati con la Student.

Tempo medio di risposta (s)	Utilizzazione
4.77252480 ± 0.04682679	3.99303450 ± 0.01229271

Tabella 5: Risultati tabellari per la configurazione 1

ID della config.	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5
0	42	962850	598499780	334435817	424484935	1068730318
1	1897151624	1565207197	2069352027	846879963	350961698	1973862481
2	441008423	1626829440	2106752724	1521090086	1313642518	1235202563
3	2039663114	1584143116	688586108	2121404701	186946556	504958512
4	1290514457	2044677693	269576733	1450020902	612424532	612942662

Tabella 6: Seed degli stream delle run

8.2 Studio 2

In questo studio si mostra la convergenza delle statistiche di un sistema con:

- La versione `2.0` del routing dinamico.
- 6 Web server, e quindi rispetta la condizione $C > E[R]$.
- Soglia di congestione $SI^{\max} = 0.4$

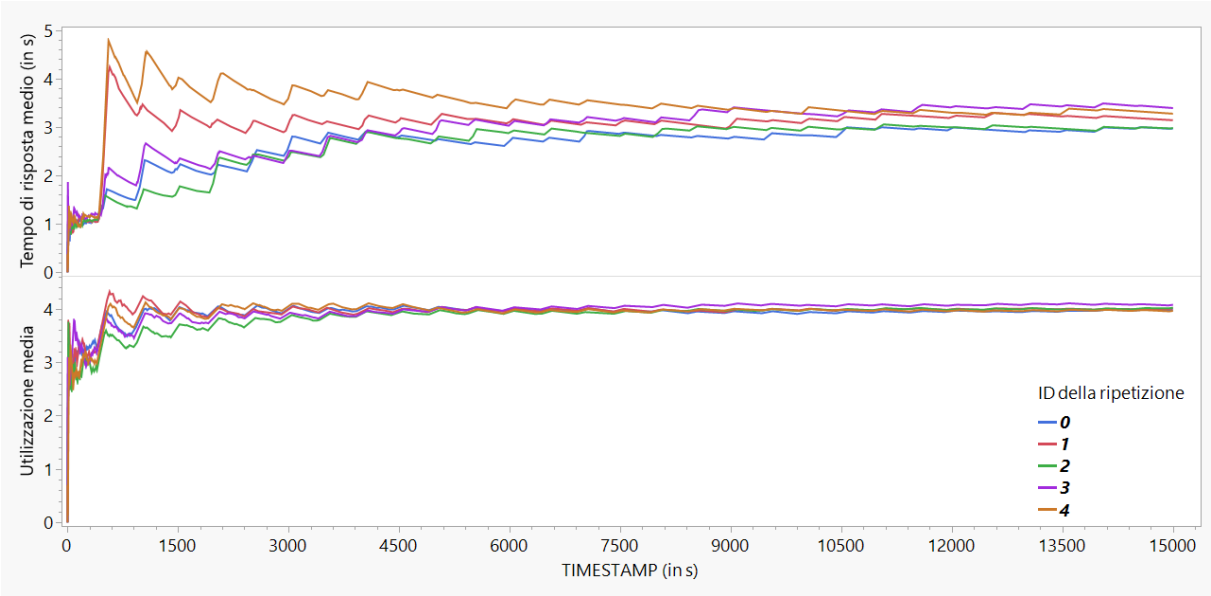

Figura 5: Studio del transitorio della configurazione 2

I risultati in formato tabellare sono stati ottenuti con 64 ripetizioni e gli intervalli di confidenza al 95% calcolati con la Student.

Tempo medio di risposta (s)	Utilizzazione
3.02994210 ± 0.00876477	3.99388852 ± 0.01229015

Tabella 7: Risultati tabellari per la configurazione 2

ID della config.	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5
0	42	962850	598499780	334435817	424484935	1068730318
1	1897151624	1565207197	2069352027	243989681	350961698	761370255
2	441008423	1626829440	2106752724	2073650376	1313642518	1726943265
3	2039663114	1584143116	688586108	1716098423	186946556	912939302
4	1290514457	2044677693	269576733	1024453771	612424532	1473032683

Tabella 8: Seed degli stream delle run

8.3 Studio 3

In questo studio si mostra la convergenza delle statistiche di un sistema con:

- La versione `2.0` del routing dinamico.
- 6 Web server, e quindi rispetta la condizione $C > E[R]$.
- Soglia di congestione $SI^{\max} = 20.0$

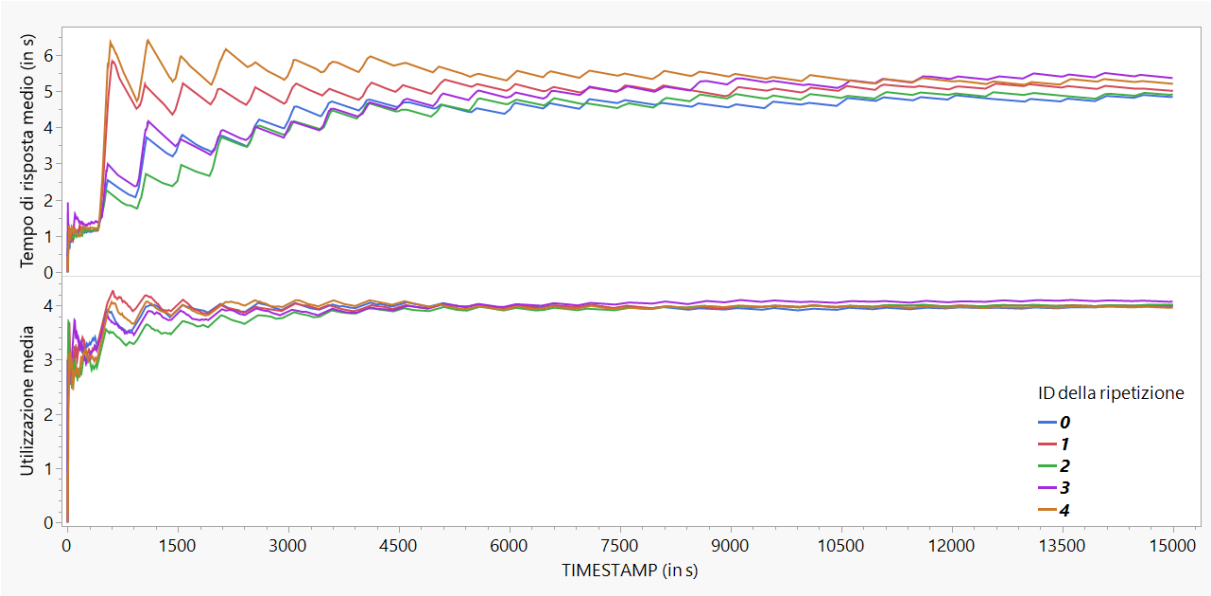

Figura 6: Studio del transitorio della configurazione 3

I risultati in formato tabellare sono stati ottenuti con 64 ripetizioni e gli intervalli di confidenza al 95% calcolati con la Student.

Tempo medio di risposta (s)	Utilizzazione
4.94235099 ± 0.01534608	3.98869265 ± 0.01226200

Tabella 9: Risultati tabellari per la configurazione 3

ID della config.	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5
0	42	962850	598499780	334435817	424484935	1068730318
1	1897151624	1565207197	2069352027	846879963	350961698	1973862481
2	441008423	1626829440	2106752724	1521090086	1313642518	1235202563
3	2039663114	1584143116	688586108	2121404701	186946556	504958512
4	1290514457	2044677693	269576733	1450020902	612424532	612942662

Tabella 10: Seed degli stream delle run

9 Studio dello stazionario: esperimenti

In questa sezione vengono descritti gli esperimenti ad orizzonte infinito: dopo aver osservato nello studio del transitorio se il sistema converge e quanto tempo impiega per farlo, ora si vuole vedere una volta arrivato a convergenza come si comporta, in modo da discriminare le configurazioni d'uso migliori.
Arrivati a questo punto, fissata la distribuzione del traffico e lo SLO da rispettare, le variabili libere da decidere sono 3:

- Il numero di Web Server.
- La soglia dello Spike Server $SI^{\max}$.
- La soluzione di routing dinamico verso lo Spike Server.

Gli esperimenti effettuati possono essere logicamente suddivisi in 3 gruppi, ma tutti hanno in comune alcuni parametri di configurazione:

- Seed pari a 42.
- Lo Spike Server è acceso.
- La capacità dei Web Server 1 $\frac{\text{operazioni}}{s}$.
- Tempi di interarrivo iperesponenziali di media 0.25 e cv pari a 4.
- Domande delle richieste iperesponenziali di media 1 e cv pari a 4.

Per ottenere i risultati sono stati utilizzati sempre 128 batch di dimensione 512, il cui tempo minimo per essere calcolati è maggiore del tempo che, dallo studio del transitorio, risulta necessario al sistema per convergere. Infatti, durante questi esperimenti la simulazione termina ad un tempo $t > 16328\ s$, che corrisponde al tempo in cui arriva il job che corrisponde all'ultimo campione dell'ultimo batch, ma come si può ben vedere il sistema converge molto prima.

I risultati di questi studi del transitorio sono visibili eseguendo la simulazione dopo aver impostato nel file di `config.properties`

```
experiment=base1
```

9.1 Gruppo 1: soluzione di routing dinamico

In questi esperimenti si studia quale soluzione di routing dinamico risulta la migliore. Gli esperimenti sono riportati in tabella 11, per brevità sono stati riportati solo quelli con $SI^{\max} \leq 4$. Come si può osservare in figura 7 la soluzione con la seconda versione routing dinamico ha un comportamento effettivamente migliore per tutte le soglie di $SI^{\max} \leq 6$ sperimentate, mentre da 6 in poi le performance delle due soluzioni tendono ad eguagliarsi, e come si vedrà anche negli esperimenti successivi, le soglie con cui otteniamo le performance migliori sono per $SI^{max} < 5$. Per questa ragione si sceglie, da questo punto in poi dello studio, di usare soltanto la seconda versione del routing dinamico verso lo Spike Server.

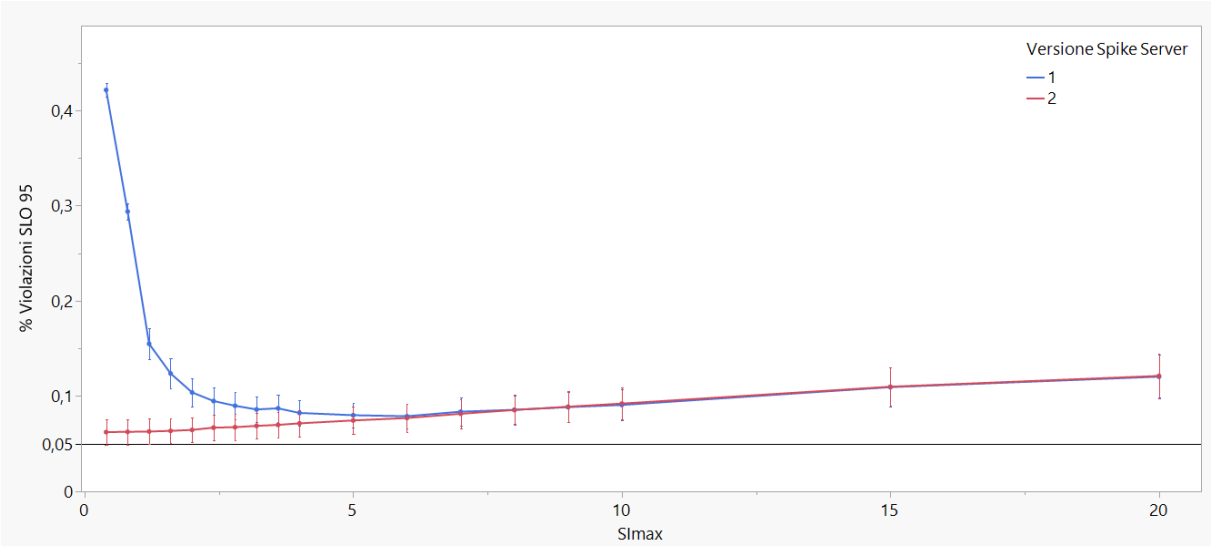

Figura 7: Esperimenti gruppo 1

Tabella 11: Esperimenti gruppo 1

Esperimento	SI^{max}	Num. WS	Ver. SS	Flutt. long-term	% Violazioni SLO
101	0.4	6	1	si	0,4261387045 ± 0,0079942135
102	0.8	6	1	si	0,2915999084 ± 0,0103507064
103	1.2	6	1	si	0,1536888685 ± 0,0171652426
104	1.6	6	1	si	0,1299458305 ± 0,0163204548
105	2	6	1	si	0,1114213779 ± 0,0153375383
106	2.4	6	1	si	0,1009536889 ± 0,0146450422
107	2.8	6	1	si	0,0970321202 ± 0,014429611
108	3.2	6	1	si	0,0888227665 ± 0,0136247873
109	3.6	6	1	si	0,089173724 ± 0,0139050925
110	4	6	1	si	0,0883039597 ± 0,0141525178
119	0.4	6	2	si	0,0670481422 ± 0,014070004
120	0.8	6	2	si	0,0663157092 ± 0,0140977578
121	1.2	6	2	si	0,0677653162 ± 0,0141102651
122	1.6	6	2	si	0,069016556 ± 0,014168087
123	2	6	2	si	0,069016556 ± 0,0140891946
124	2.4	6	2	si	0,0698252842 ± 0,0141772149
125	2.8	6	2	si	0,0712901503 ± 0,0144250562
126	3.2	6	2	si	0,0720531014 ± 0,0145898959
127	3.6	6	2	si	0,0729533837 ± 0,0144667057
128	4	6	2	si	0,0742198825 ± 0,0148943311

9.2 Gruppo 2: ricerca configurazione migliore

Continuando sulla scia degli ultimi esperimenti, si cerca ora che configurazione della coppia di parametri (SI^{max}, Num. WS) è la migliore. Gli esperimenti sono stati descritti in tabella 12, dove per brevità sono stati riportati solo gli esperimenti con un numero di Web Server pari a 7.
Come si può osservare dalla figura 8, le configurazioni migliori sono sempre ottenute per valori di SI^{max} più piccole, ovvero in cui il sistema fa un utilizzo più preventivo dello Spike Server, e nonostante la capacità teorica richista $E[R]$ sia di 4 $\frac{operazioni}{s}$, servono comunque almeno 7 Web Server per rispettare lo SLO al 95%, configurazione nella quale è allocata in totale una capacità di 8 $\frac{operazioni}{s}$, ovvero quella dei 7 Web Server più quella dello Spike Server.

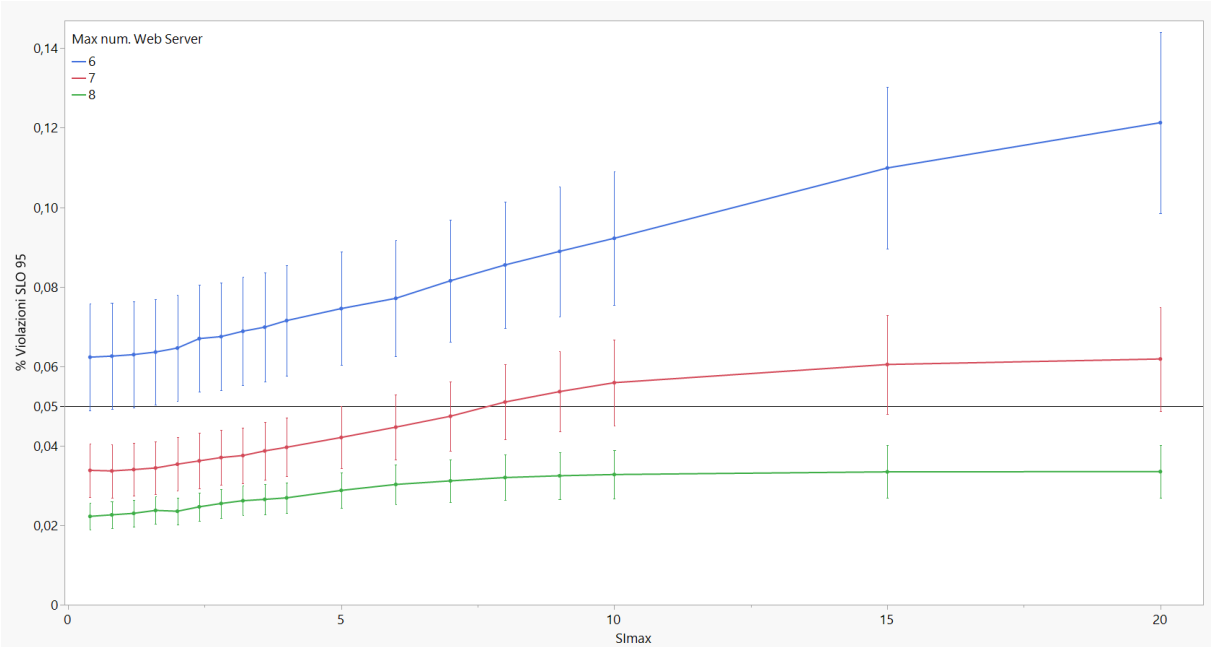

Figura 8: Esperimenti gruppo 2

Tabella 12: Esperimenti gruppo 2

Esperimento	SI^{max}	Num. WS	Ver. SS	Flutt. long-term	% Violazioni SLO
219	0.4	7	2	si	0,0353093767 ± 0,0066519662
220	0.8	7	2	si	0,0355840391 ± 0,0069491433
221	1.2	7	2	si	0,036255436 ± 0,006828935
222	1.6	7	2	si	0,0369726101 ± 0,0068867528
223	2	7	2	si	0,0371557183 ± 0,0068086837
224	2.4	7	2	si	0,0380254826 ± 0,0070701496
225	2.8	7	2	si	0,0388494697 ± 0,0069936238
226	3.2	7	2	si	0,039703975 ± 0,0072575992
227	3.6	7	2	si	0,0408484016 ± 0,0073113347
228	4	7	2	si	0,0417029068 ± 0,00738768
229	5	7	2	si	0,0440833143 ± 0,0078784372
230	6	7	2	si	0,0477912566 ± 0,0085944189
231	7	7	2	si	0,0509193561 ± 0,0092421724
232	8	7	2	si	0,0534828717 ± 0,0097427656
233	9	7	2	si	0,0560463874 ± 0,0103988209
234	10	7	2	si	0,0587624933 ± 0,0111658193
235	15	7	2	si	0,0655680171 ± 0,0135334687
236	20	7	2	si	0,0666971847 ± 0,0140933617

In questa fase degli esperimenti si sottopone anche un'interessante osservazione: usando gli stessi dati degli esperimenti del grafico in figura 8, e raggruppandoli sulle ascisse in base al numero di Web Server attivi in ordine decrescente, si ottiene il grafico in figura 9.
Come si può osservare da questo grafico, in una configurazione con $(n+1)$ server in cui $SI^{\max} \to \infty$, ovvero in cui si fa sempre meno utilizzo dello Spike Server, si ottengono prestazioni che tendono a quelle di una configurazione con n server in cui $SI^{\max} \to 0$, ovvero in cui lo Spike Server viene sempre più utilizzato. Questo mostra il legame tra la la percentuale di violazioni allo SLO 95 con la capacità totale allocata al secondo e $SI^{\max}$, che sembra una funzione monotona crescente e suggerisce l'esistenza di una soluzione ottima.

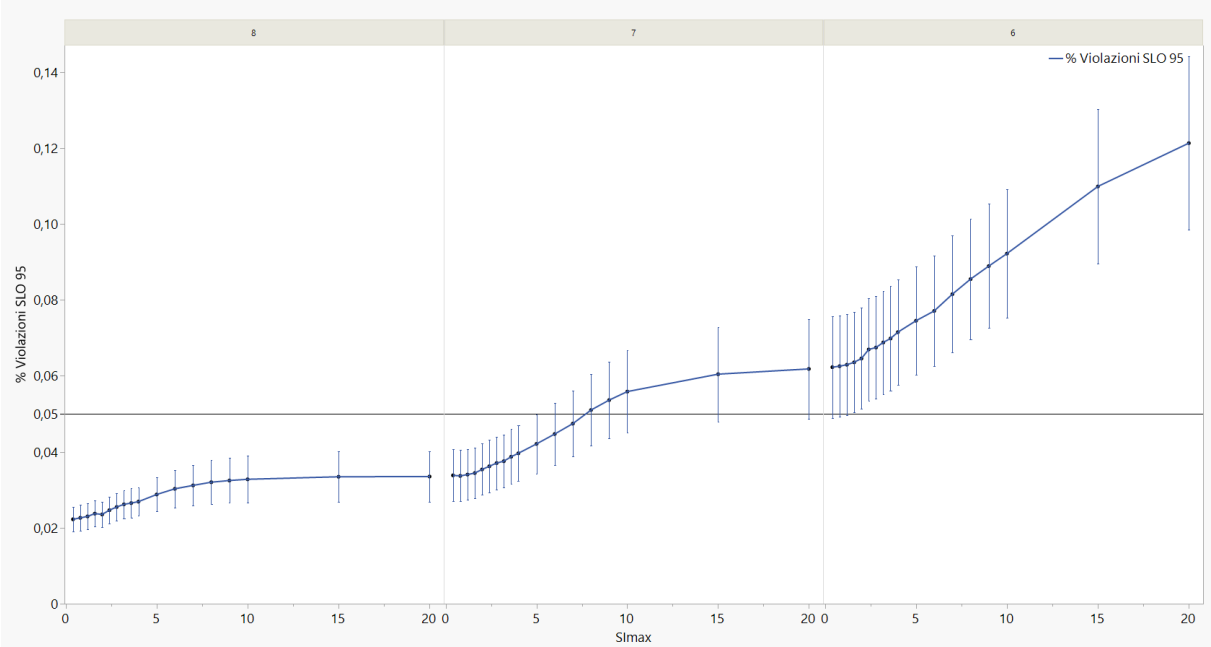

Figura 9: Esperimenti gruppo 2, raggruppati da sinistra verso destra con un numero di server pari a 8, 7 e 6

9.3 Gruppo 3: necessità di sovra-dimensionare il sistema

In questa sezione si vuole mostrare che le fluttuazioni comportano la necessità di sovradimensionare il sistema. A tale scopo sono stati effettuati 2 esperimenti descritti in tabella 13, e raffigurati in figura 10.

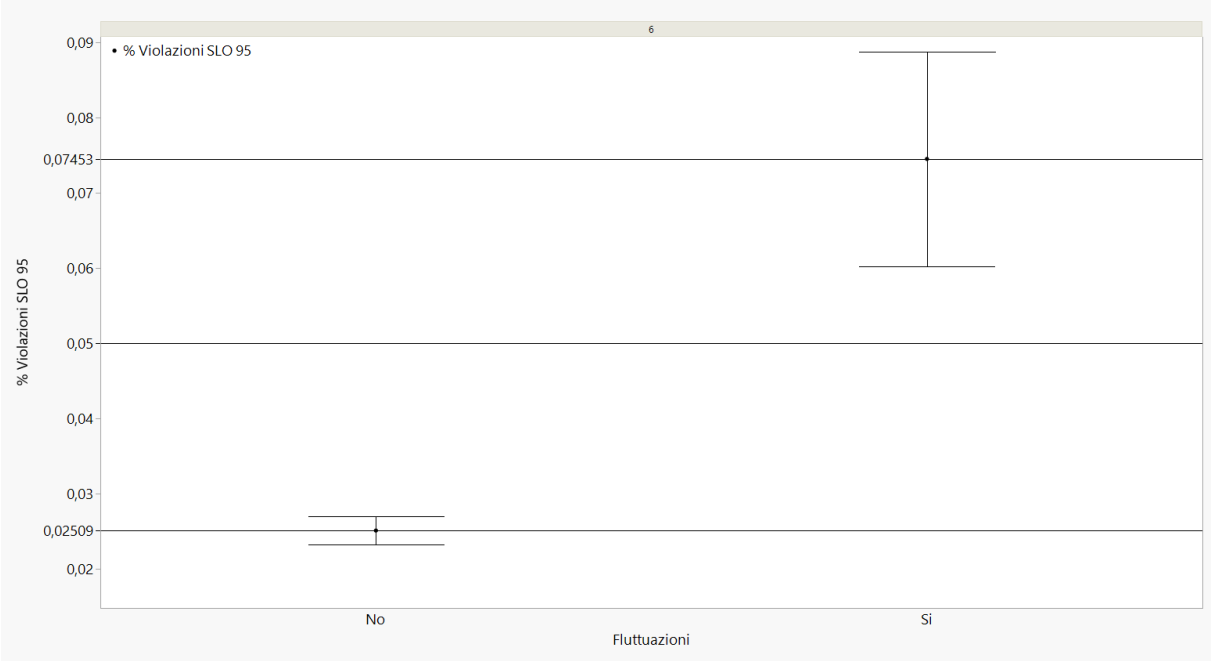

Figura 10: Esperimenti gruppo 3

Tabella 13: Esperimenti gruppo 3

Esperimento	SI^{max}	Num. WS	Ver. SS	Flutt. long-term	% Violazioni SLO
301	5	6	2	no	0.02508583 ± 0.00191551
302	5	6	2	si	0.07453177 ± 0.01514887

Come si può osservare dai grafici in figura 11 e 12, senza fluttuazioni il sistema rispetterebbe gli SLO posti in obiettivo, ma la presenza delle fluttuazioni comporta, proprio come ci si aspetterebbe, un picco non solo di arrivi ma anche di violazioni, picco che può essere trattato soltanto tramite un aumento della capacità computazionale totale del sistema.

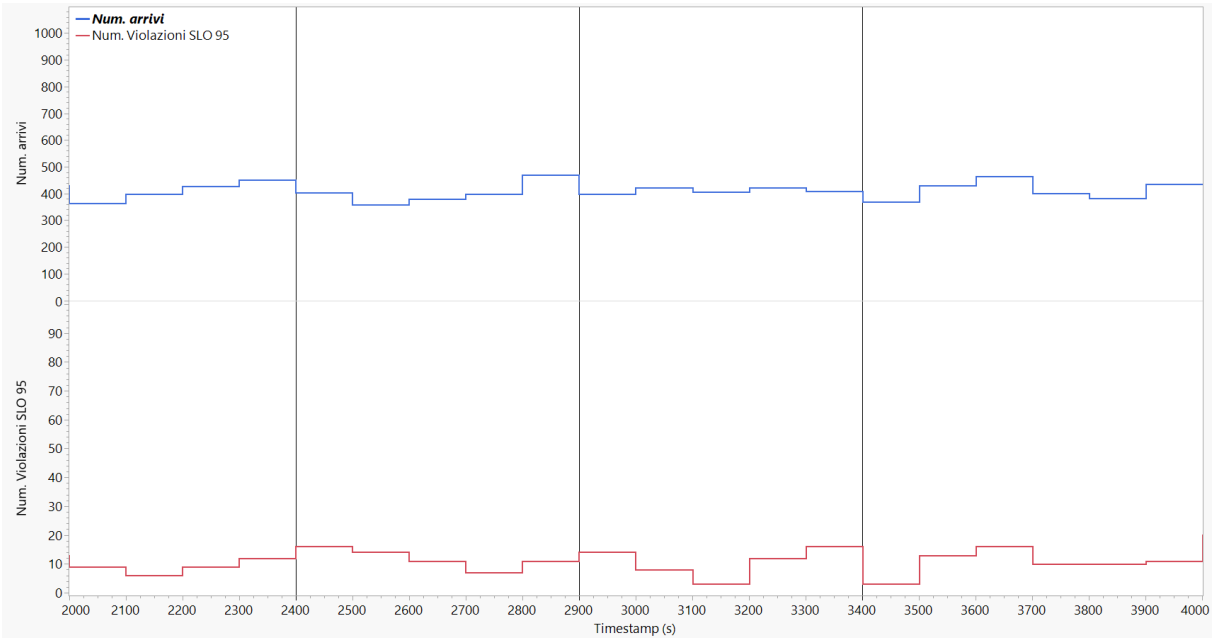

Figura 11: Esperimento 301

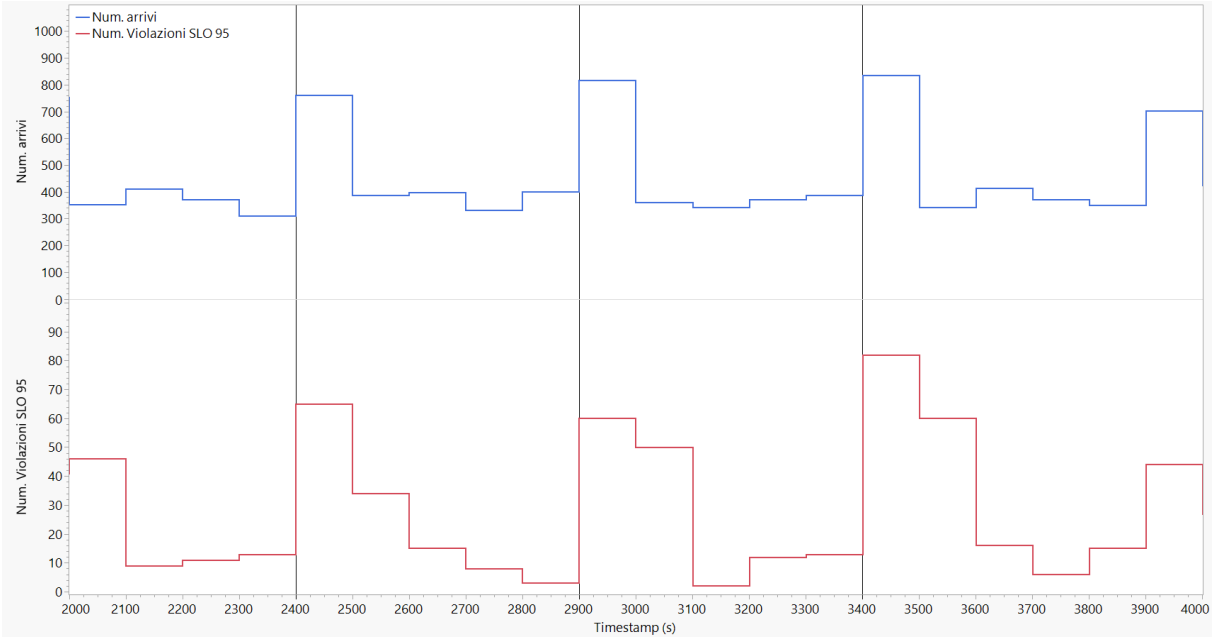

Figura 12: Esperimento 302

10 Introduzione al modello migliorativo

10.1 Oggetto di studio

Nel modello base si considera un cluster di server progettato per gestire le fluttuazioni di carico a breve termine (short-term). In situazioni di congestione, piuttosto che inviare immediatamente le richieste ai Web Server, queste vengono deviate verso uno Spike Server, il quale esegue le richieste al loro posto permettendone il decongestionamento. Tuttavia, **quando le variazioni di carico sono persistenti nel tempo (long-term) questa soluzione risulta poco efficiente**.
Infatti, come visto negli esperimenti precedenti, per rispettare gli SLO imposti vi è la necessità di sovradimensionare il sistema, e questo comporta un sottoutilizzo delle risorse allocate nei periodi di bassa intensità di traffico.
Per tale motivo, è stata implementata una strategia di scaling orizzontale che permette di aumentare o ridurre dinamicamente il numero di Web Server attivi in base al carico effettivo. Tale meccanismo si basa sul monitoraggio di un indicatore di performance, che verrà spiegato nelle sezioni successive.
L'obiettivo è analizzare quale metrica risulti più affidabile e rappresentativa per guidare le politiche di scaling orizzontale, bilanciando al meglio l'utilizzo delle risorse e la qualità del servizio offerto.

11 Modello concettuale migliorativo

11.1 Componenti

11.1.1 Web servers

La presenza dell'autoscaler può far accendere o spegnere un server, ma quando si accende un server spento questo non diventa subito acceso per via del tempo di booting, e quando si decide di spegnere un server con dei job ancora in esecuzione questo non può essere subito spento, per cui è importante definire i vari stati in cui un server può trovarsi.
Per questo motivo si definisco i possibili "stati di attivazione" in cui un Web Server si può trovare:

- `ACTIVE`: il WS è completamente attivo.
  - Le risorse del WS sono allocate.
  - Il WS può essere scelto come risorsa per evadere una richiesta.
  - Il WS può avere o meno delle richieste in fase di evasione.
- `TO_BE_REMOVED`: il WS è attivo, ma in fase di spegnimento.
  - Le risorse del WS sono ancora allocate.
  - Il WS **non** può essere scelto come risorsa per evadere una richiesta.
  - Il WS ha ancora delle richieste in fase di evasione.
  - Portare un WS da questo stato allo stato `ACTIVE` richiede un tempo trascurabile.
- `REMOVED`: il WS è completamente spento.
  - Le risorse del WS non sono allocate.
  - Il WS non può essere scelto come risorsa per evadere una richiesta.
  - Il WS non ha richieste in fase di evasione.
  - Portare un WS da questo stato allo stato `ACTIVE` richiede un tempo non trascurabile.
- `TO_BE_ACTIVE`: come `REMOVED` ma il WS è in fase di accensione.

11.1.2 Load controller

Nel modello migliorativo il Load controller viene esteso per prendere le decisioni di scaling. In particolare, oltre alle attività fatte per il modello base si occupa anche di monitorare 1 o più indicatori di performance e decidere se applicare azioni di scaling orizzontale per i Web Server.

11.2 Eventi del sistema

Il modello migliorativo introduce lo scaling orizzontale che viene modellato attraverso i seguenti eventi:

- Scaling out: l'aumento del numero di server non è un'operazione immediata ma richiede un tempo non trascurabile. Quindi avremo un evento per la richiesta di scaling out e un evento che materializza l'operazione di scaling out.
- Scaling in: viene scelto il Web Server da rimuovere, il quale verrà effettivamente rimosso solo una volta che avrà smaltito le rimanenti richieste.

12 Modello delle specifiche migliorativo

12.1 Inizializzazione del sistema

All'inizio della simulazione un solo server è acceso, ovvero allo stato `ACTIVE`, mentre tutti gli altri sono impostati allo stato `REMOVED`.

12.2 Modellazione auto-scaling orizzontale

12.2.1 Indicatori di scaling

Per scegliere l'azione di autoscaling, il Load Controller può monitorare due indicatori diversi:

- Basato sul tempo di risposta.
  In particolare si calcola la media delle finestre scorrevoli di ciascun Web Server, e il valore dell'indicatore è calcolato aggregando questi risultati in una media finale. L'indicatore viene ricalcolato ad ogni completamento.
- Basato sul numero di job nel sistema (considerando sia il Web Server che lo Spike Server).
  In particolare, il valore dell'indicatore viene calcolato semplicemente come la somma dei job presenti nei Web Server `ACTIVE` e `TO_BE_REMOVED` e dei job presenti nello Spike Server. L'indicatore viene ricalcolato ad ogni arrivo e completamento.

12.2.2 Azioni di scaling

Le azioni di scaling vengono prese secondo il seguente approccio:

1. Utilizzare la soglia impostata sull'indicatore scelto, indicata come $\text{scaling}_{\text{thr}}$, per determinare se c'è la necessità di una azione di scaling.
   - Se l'indicatore è sul tempo di riposta: l'azione di scale-out viene eseguita se l'indicatore è superiore al 150% della soglia impostata, mentre l'azione di scale-in viene eseguita se l'indicatore è inferiore al 50% della soglia impostata.
   - Se l'indicatore è sul numero di job: si determina il numero di Web Server stimato come $\text{WS}_{\text{expected}} = \lceil \frac{\text{num job nel sist.}}{\text{scaling}_{\text{thr}}} \rceil$. L'azione di scale-out viene eseguita se l'attuale numero di Web Server attivi è inferiore al numero di Web Server stimato, mentre l'azione di scale-in viene eseguita se il numero di Web Server attivi è superiore al numero di Web Server stimato.
2. Se viene stabilito di attivare un'azione di scaling, si controlla che l'azione sia possibile:
   - Se l'azione è di scale-out devono essere presenti dei Web Server allocabili, cioè in stato `TO_BE_REMOVED` o `REMOVED`.
   - Se l'azione è di scale-in devono essere presenti più di 2 Web Server allocati, poiché almeno un Web Server deve rimanere sempre allocato.
3. Si stabilisce il Web Server su cui applicare l'azione di scaling con le seguenti preferenze:
   - Se l'azione è di scale-out, si preferiscono i Web Server nello stato `TO_BE_REMOVED` in modo da non soffrire dei tempi di accensione.
   - Se l'azione è di scale-in, si preferiscono i Web Server nello stato `TO_BE_ACTIVE` in modo da non dover aspettare che un server si scarichi.

12.2.3 Tempi di accensione

Se il server scelto per l'operazione di scaling è completamente spento, cioè nello stato `REMOVED`, la sua accensione richiederà un certo tempo.

Il tempo di accensione è stato modellato utilizzando una variabile aleatoria Normale di media 5 s e varianza 0.5 s^2. Il tempo è modellato per tutti i web server con questa stessa distribuzione. Ci si aspetta infatti una bassa variabilità poichè, assumendo che i Web Server siano omogenei, il tempo di accensione mediamente sarà lo stesso.

13 Modello computazionale migliorativo

13.1 La gestione degli eventi

La presenza dell'autoscaling richiede la gestione di ulteriori eventi rispetto al modello base:

- Scaling in.
- Scaling out.
- Scaling out request.

La richiesta di scaling out modella il fatto che l'accensione di un web server non è immediata.

13.1.1 Evento di arrivo

Rispetto al modello base, all'occorrere di un arrivo è necessario aggiornare gli indicatori che vengono utilizzati per prendere le decisioni di scaling, e in caso, schedularle.

13.1.2 Evento di completamento

Rispetto al modello base, all'occorrere di un completamento è necessario aggiornare gli indicatori che vengono utilizzati per prendere le decisioni di scaling, e in caso, schedularle.
Dopo aver calcolato l'avanzamento dei job, se un Web Server che si trova nello stato `TO_BE_REMOVED` risulta essere senza job assegnati, allora questo viene portato allo stato `REMOVED`.

13.1.3 La funzione che pianifica lo scaling

Nelle sezioni precedenti abbiamo visto che viene utilizzata la funzione `planScaling()`. Quest'ultima nel caso in cui si è deciso di utilizzare il modello migliorativo, ovvero è stata configurata una soglia per lo scaling `SCALING_OUT_THRESHOLD` finita, pianifica un evento un evento di scaling in oppure fa una richiesta di scaling out.

```
private void planScaling(SystemState s, double endTs, double scalingIndicator){
    IServerInfrastructure servers = s.getServers();
    if (SCALING_OUT_THRESHOLD != INFINITY){
        ...
        int activatedServer = servers.getNumWebServersByState(ServerState.ACTIVE) +
                              servers.getNumWebServersByState(ServerState.TO_BE_ACTIVE);

        scalingOutPossible = activatedServer < MAX_NUM_SERVERS;
        scalingInPossible = servers.getNumWebServersByState(ServerState.ACTIVE) > 1;

        if (SCALING_INDICATOR_TYPE.equals("r0")) {
            scalingOutCondition = scalingIndicator > SCALING_OUT_THRESHOLD * 1.5;
            scalingInCondition = scalingIndicator < SCALING_OUT_THRESHOLD * 0.5;
        } else if (SCALING_INDICATOR_TYPE.equals("jobs")) {
            int expectedServers = (int) (Math.ceil(scalingIndicator / SCALING_OUT_THRESHOLD));
            scalingOutCondition = activatedServer < expectedServers;
            scalingInCondition = activatedServer > expectedServers;
        }

        if (scalingOutCondition && scalingOutPossible)
            s.addEvent(new ScalingOutReqEvent(endTs));
        else if (scalingInCondition && scalingInPossible)
            s.addEvent(new ScalingInEvent(endTs));
    }

}
```

Listing 13: Codice che implementa la pianificazione di un evento di scaling.

13.1.4 Evento richiesta di Scaling Out

Dato che il tempo per accendere un Web Server non è trascurabile, per le operazioni di scaling out si deve fare una richiesta. Si dovrà quindi cercare un server che rispetta una delle seguenti proprietà:

- Si cerca un server che è ancora attivo ma da rimuovere. Ovvero, un server nello stato `TO_BE_REMOVED`.
- Se non ci sono server in fase di disattivazione, si sceglie quello con indice minimo che si trova nello stato `REMOVED`.

Tale server, se disponibile, verrà portato nello stato `TO_BE_ACTIVE` e verrà settato come istante di attivazione un tempo pari alla somma tra:

- L'istante in cui è stata triggerata la richiesta
- Un tempo campionato da una distribuzione normale che modella il tempo di accensione di un server.

La necessità di effettuare la richiesta prima dell'azione di scaling out effettiva è dovuta al fatto che il tempo di attivazione dei server non è deterministico. Quindi, se due richieste di scaling-out sono molto vicine, più server possono trovarsi nello stato `TO_BE_ACTIVE`, ed è possibile che l'ordine delle richieste non corrisponda all'ordine delle effettive accensioni. Per cui, dopo aver calcolato il momento dell'accensione del server da allocare, viene ricalcolato il server che ha il tempo d'accensione più vicino in termini temporali e si schedula un evento di scaling out utilizzando come timestamp il tempo trovato.

```java
@Override
public void visit(SystemState s, ScalingOutReqEvent event) {
    IServerInfrastructure servers = s.getServers();
    double endTs = event.getTimestamp();

    /* Generate time for turn on a web sever */
    double turnOnTime = s.getTurnOnVA().gen();

    // From request to effective scale out
    var serverTarget = servers.requestScaleOut(endTs, turnOnTime);

    /* Find the earliest WS to make active: it needs to be done
       in case no scaling out event is already scheduled */
    WebServer nextScaleOut = servers.findNextScaleOut();
    s.addEvent(new ScalingOutEvent(nextScaleOut.getActivationTimestamp(), nextScaleOut));

    // Schedule to INFINITY the next request
    s.addEvent(new ScalingOutReqEvent(INFINITY));

    /* Update the current system clock */
    s.setCurrent(endTs);
}
```

Listing 14: Codice che implementa evento di richiesta di scaling out

13.1.5 Evento di Scaling out

Una volta preso il clock corrente e processato l'avanzamento dei job, il server per cui era stato richiesto di scalare viene effettivamente portato nello stato `ACTIVE`. Dopo di che viene cercato il prossimo server per cui generare un evento di scaling out, se ne esiste uno in stato `TO_BE_ACTIVE`.

```java
@Override
public void visit(SystemState s, ScalingOutEvent event) throws IllegalLifeException {
    IServerInfrastructure servers = s.getServers();

    /* Get the current clock and the one of this arrival */
    double startTs = s.getCurrent();
    double endTs = event.getTimestamp();

    /* Advance job execution */
    servers.computeJobsAdvancement(startTs, endTs, false);

    /* Set server to be effectively active */
    servers.scaleOut(endTs, event.getTarget());

    /* Find the earliest WS to make active */
    WebServer nextScaleOut = servers.findNextScaleOut();
    double nextActivationTS = nextScaleOut == null ? INFINITY : nextScaleOut.getActivationTimestamp();
    s.addEvent(new ScalingOutEvent(nextActivationTS, nextScaleOut));

    /* Update the current system clock */
    s.setCurrent(endTs);
}
```

Listing 15: Codice che implementa evento di richiesta di scaling out

13.1.6 Evento di Scaling In

Quando si verifica un evento di scaling in si cerca un server tale che:

- Si trova nello stato `TO_BE_ACTIVE`: dunque un server che deve essere attivato ma ancora non è attivo.
- Se nessun server è ancora in fase di accensione, se ne cerca uno `ACTIVE`.

Il server individuato dovrà essere portato nello stato `TO_BE_REMOVED` se ancora possiede dei job da eseguire, altrimenti è portato direttamente nello stato `REMOVED`.

```java
@Override
public void visit(SystemState s, ScalingInEvent event) {
    IServerInfrastructure servers = s.getServers();
    double endTs = event.getTimestamp();

    servers.scaleIn(endTs);

    s.addEvent(new ScalingInEvent(INFINITY));

    s.setCurrent(endTs);
}
```

Listing 16: Codice che implementa evento di richiesta di scaling out

13.1.7 La variabile aleatoria Normale

La variabile aleatoria normale è stata implementata utilizzando la **trasformazione di Box-Muller** che ci permette di ottenere una distribuzioni normale standard a partire da due distribuzioni uniformi.
Per ottenere una variabile aleatoria normale con media e varianza generica si è applicata la formula:

$$Z' = \mu + Z\sigma$$

Dove: Z è una variabile aleatoria normale standard, invece μ e σ sono la media e la deviazione standard che vogliamo applicare in modo che Z' sia una variabile aleatoria normale con parametri μ e σ^2.

14 Validazione modello migliorativo

Questa volta si vuole rispondere alla domanda:

1. L'autoscaler alloca i server in corrispondenza dei picchi di carico?

I risultati di questi studi sono visibili eseguendo la simulazione dopo aver impostato nel file di `config.properties`

```
experiment=val_adv
```

14.1 Domanda 1

L'autoscaler alloca i server in corrispondenza dei picchi di carico?
Per validare questa condizione configuriamo il sistema nel seguente modo:

- seed pari a 42.
- lo Spike Server è acceso con $SI^{max} = 3$.
- 10 Web Server di capacità 1 $\frac{\text{operazioni}}{s}$.
- tempi di interarrivo iperesponenziali di media 0.25 e cv pari a 4
- domande delle richieste iperesponenziali di media 1 e cv pari a 4
- fluttuazioni long-term presenti

Un estratto del comportamento aggiornato del sistema è visualizzabile in figura 13: il grafico è stato prodotto aggregando le informazioni per intervalli di 100 s, e ogni intervallo contiene il numero di arrivi presenti in esso e il numero medio di Web Server attivi.

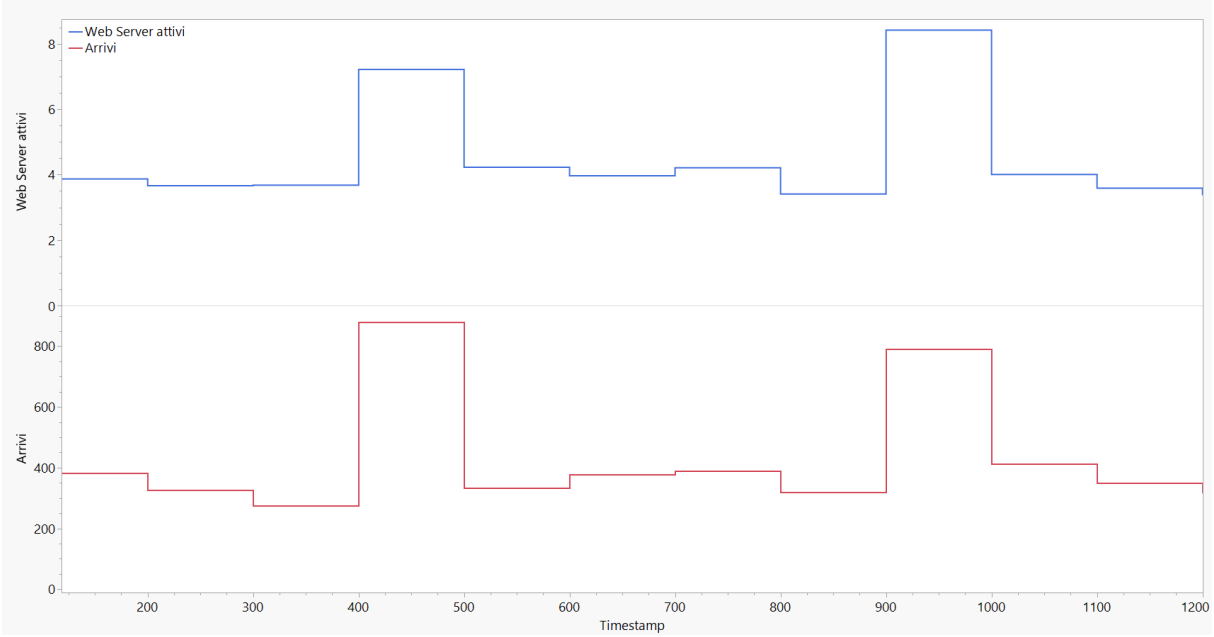

Figura 13: Numero web server attivi in corrispondenza dei picchi di arrivo

Come si può vedere, in corrispondenza dei periodi di picco, il numero di server attivi aumenta, e quando queste fluttuazioni terminano il sistema capisce che può deallocare le risorse.

15 Studio del transitorio per il modello migliorativo

In questa fase si mostrano lo studio del transitorio di alcune configurazioni, per assicurare che le configurazioni utilizzate negli esperimenti presenti nella sezione successiva corrispondano a simulazioni le cui statistiche convergono.

In questo caso, la metrica da comparare alla richiesta totale per secondo è la:

- $C_{\max}$ Capacità massima allocabile totale per secondo: espressa in $\frac{\text{operazioni}}{s}$, è la somma delle capacità dei server allocabili (sia Web Server, sia Spike Server).

$$C_{\max} = \sum_{\text{Server allocabili}} C_i$$

In generale, dagli studi condotti sembra emergere che le statistiche del sistema convergono se il sistema ha una capacità massima allocabile per secondo maggiore della richiesta per secondo.

$$\text{sistema converge} \iff C_{\max} > E[R]$$

Inoltre, è risultato che le statistiche del sistema convergono per entrambi gli indicatori di scaling orizzontale (sul tempo di risposta e sul numero di job) e su qualsiasi soglia massima.

In tutti gli studi vengono fissati i seguenti parametri:

- Distribuzioni iperesponenziali di arrivi e richieste per job, con i seguenti parametri.

$$\begin{aligned} E[\lambda] &= 4 \ \frac{\text{arrivi}}{s} \\ E[Z] &= 1 \ \text{operazioni} \\ &\Downarrow \\ E[R] &= 4 \ \frac{\text{operazioni}}{s} \end{aligned}$$

$$\begin{aligned} CV_{\text{arrivi}} &= 4.0 \\ CV_{\text{richieste}} &= 4.0 \end{aligned}$$

- Fluttuazioni long-term con periodo di 500 s e i seguenti parametri:

$$\begin{aligned} \lambda_{\text{slow}} &= 3 \ \frac{\text{arrivi}}{s} \ \text{per } 400 \ s \\ \lambda_{\text{fast}} &= 8 \ \frac{\text{arrivi}}{s} \ \text{per } 100 \ s \end{aligned}$$

I risultati di questi studi sono visibili eseguendo la simulazione dopo aver impostato nel file di `config.properties`

```
experiment=trans_adv
```

15.1 Studio 1

In questo studio si mostra la convergenza delle statistiche di un sistema con:

- Massimo 10 Web server attivi, e quindi rispetta la condizione $C_{\max} > E[R]$.
- Soglia di congestione $SI^{\max} = 2.0$
- Indicatore di scaling orizzontale basato sul tempo di risposta, con lunghezza della finestra mobile di 8.
- Soglia di scaling orizzontale $\text{scaling}_{\text{thr}} = 3.0\ s$

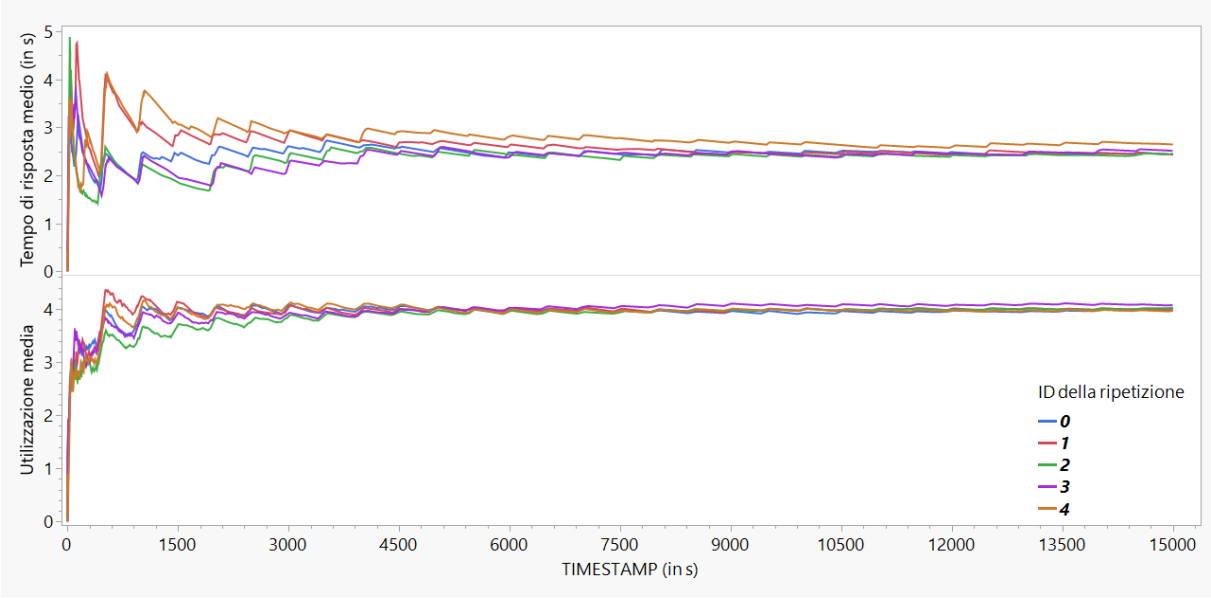

Figura 14: Studio del transitorio della configurazione 1 avanzata

I risultati in formato tabellare sono stati ottenuti con 64 ripetizioni e gli intervalli di confidenza al 95% calcolati con la Student.

Tempo medio di risposta (s)	Utilizzazione
2.43201073 ± 0.02881553	3.99832909 ± 0.01221449

Tabella 14: Risultati tabellari per la configurazione 1 avanzata

ID della config.	Stream 0	Stream 1	Stream 2	Stream 3
0	42	962850	598499780	334435817
1	1897151624	1565207197	2069352027	243989681
2	441008423	1626829440	2106752724	2073650376
3	2039663114	1584143116	688586108	1716098423
4	1290514457	2044677693	269576733	1450020902

Tabella 15: Prima parte dei seed degli stream delle run

ID della config.	Stream 4	Stream 5	Stream 6	Stream 7
0	424484935	1068730318	1611527	437034476
1	350961698	761370255	884907689	1378240763
2	1313642518	1726943265	1092902644	101404151
3	186946556	912939302	553593501	1650140302
4	612424532	612942662	1710988213	615970570

Tabella 16: Seconda parte dei seed degli stream delle run

15.2 Studio 2

In questo studio si mostra la convergenza delle statistiche di un sistema con:

- Massimo 10 Web server attivi, e quindi rispetta la condizione $C_{\max} > E[R]$.
- Soglia di congestione $SI^{\max} = 2.0$
- Indicatore di scaling orizzontale basato sul numero di job nel sistema.
- Soglia di scaling orizzontale $\text{scaling}_{\text{thr}} = 3.0\ \frac{\text{job}}{\text{server attivo}}$.

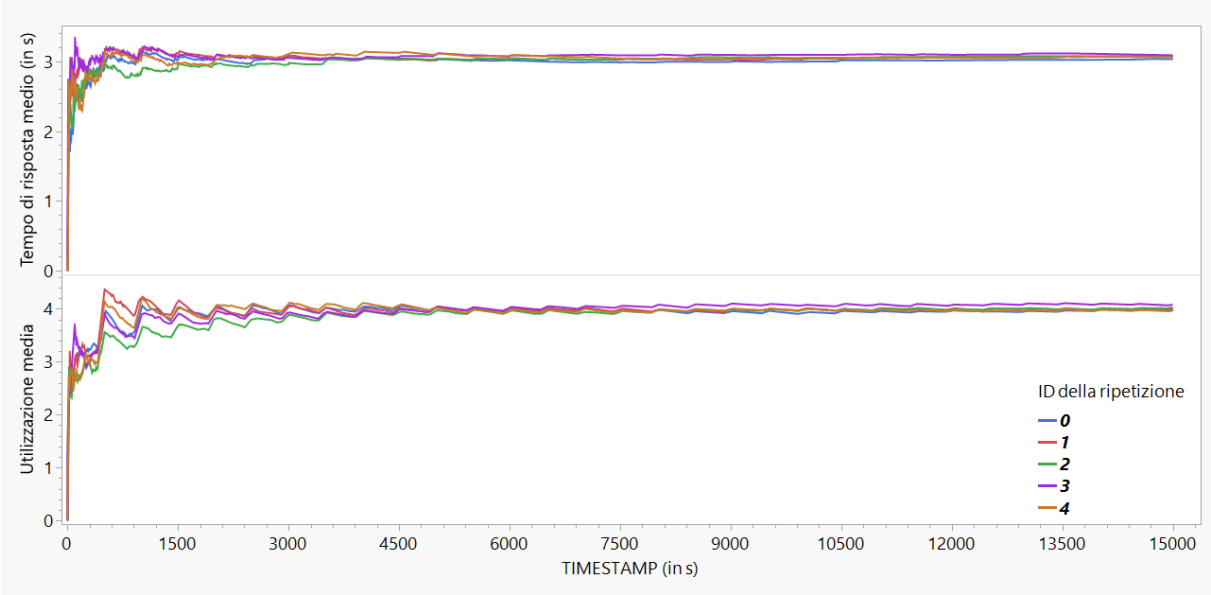

Figura 15: Studio del transitorio della configurazione 2 avanzata

I risultati in formato tabellare sono stati ottenuti con 64 ripetizioni e gli intervalli di confidenza al 95% calcolati con la Student.

Tempo medio di risposta (s)	Utilizzazione
3.07463496 ± 0.00785682	3.99783285 ± 0.01217492

Tabella 17: Risultati tabellari per la configurazione 1 avanzata

ID della config.	Stream 0	Stream 1	Stream 2	Stream 3
0	42	962850	598499780	334435817
1	1897151624	1565207197	2069352027	243989681
2	441008423	1626829440	2106752724	2073650376
3	2039663114	1584143116	688586108	1716098423
4	1290514457	2044677693	269576733	1302040472

Tabella 18: Prima parte dei seed degli stream delle run

ID della config.	Stream 4	Stream 5	Stream 6	Stream 7
0	424484935	1068730318	1611527	437034476
1	350961698	761370255	1457188163	1930507690
2	1313642518	1726943265	310933609	654761932
3	370919982	2128610402	1946406860	962048134
4	612424532	1577088923	1891116604	448281064

Tabella 19: Seconda parte dei seed degli stream delle run

15.3 Studio 3

In questo studio si mostra la convergenza delle statistiche di un sistema con:

- Massimo 6 Web server attivi, e quindi rispetta la condizione $C_{\max} > E[R]$.
- Soglia di congestione $SI^{\max} = 2.0$
- Indicatore di scaling orizzontale basato sul numero di job nel sistema.
- Soglia di scaling orizzontale $\text{scaling}_{\text{thr}} = 5.0 \ \frac{\text{job}}{\text{server attivo}}$.

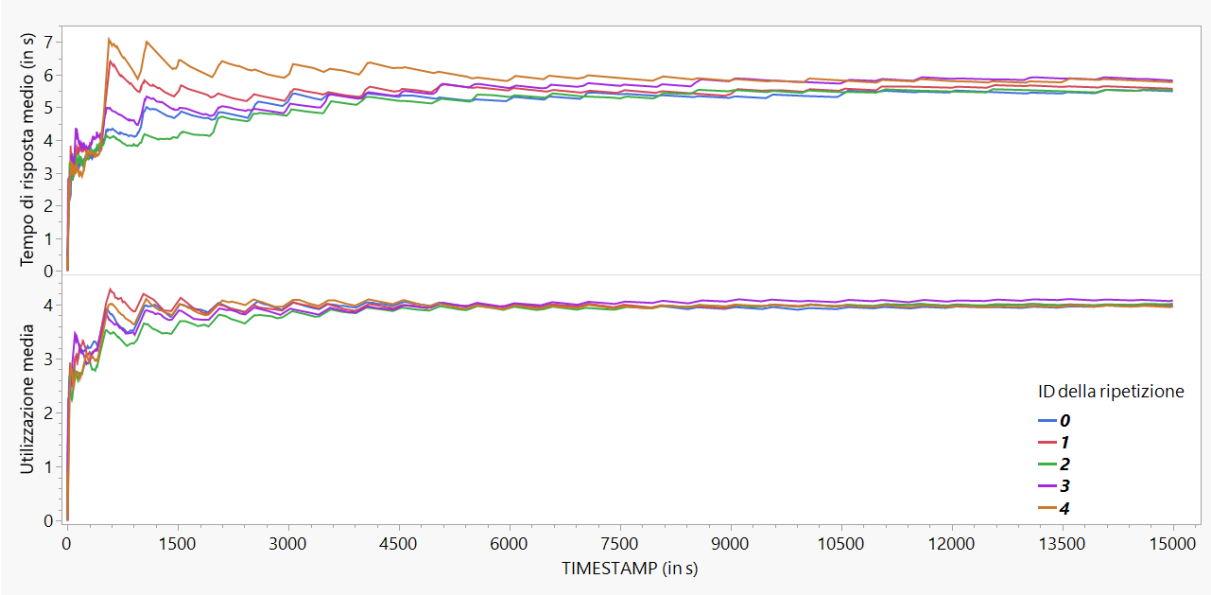

Figura 16: Studio del transitorio della configurazione 3 avanzata

I risultati in formato tabellare sono stati ottenuti con 64 ripetizioni e gli intervalli di confidenza al 95% calcolati con la Student.

Tempo medio di risposta (s)	Utilizzazione
5.53916156 ± 0.04589572	3.99130603 ± 0.01233910

Tabella 20: Risultati tabellari per la configurazione 1 avanzata

ID della config.	Stream 0	Stream 1	Stream 2	Stream 3
0	42	962850	598499780	334435817
1	1897151624	1565207197	2069352027	846879963
2	441008423	1626829440	2106752724	1521090086
3	2039663114	1584143116	688586108	1716098423
4	1290514457	2044677693	269576733	1450020902

Tabella 21: Prima parte dei seed degli stream delle run

ID della config.	Stream 4	Stream 5	Stream 6	Stream 7
0	424484935	1068730318	1611527	437034476
1	350961698	1973862481	1329735753	661768360
2	1313642518	1235202563	2137426534	1368918351
3	186946556	912939302	1303616080	1016202348
4	612424532	612942662	1038712371	1182427239

Tabella 22: Seconda parte dei seed degli stream delle run

16 Esperimenti per il modello migliorativo

In questa sezione vengono descritti gli esperimenti ad orizzonte infinito: dopo aver osservato nello studio del transitorio se il sistema converge e quanto tempo il sistema impiega per farlo, ora si vuole vedere una volta arrivato a convergenza come si comporta, in modo da discriminare le configurazioni d'uso migliori.

Fissata la distribuzione del traffico e lo SLO da rispettare, le variabili libere da decidere sono 4:

- Il numero massimo di Web Server.
- La soglia dello Spike Server $SI^{\max}$.
- Il tipo di indicatore per l'autoscaling.
- La soglia dell'indicatore dell'autoscaling.

Gli esperimenti effettuati possono essere logicamente suddivisi in 2 gruppi, ma tutti hanno in comune alcuni parametri di configurazione:

- Seed pari a 42.
- Lo Spike Server è acceso.
- Seconda soluzione di routing dinamico.
- La capacità dei Web Server 1 $\frac{\text{operazioni}}{s}$.
- Tempi di interarrivo iperesponenziali di media 0.25 e cv pari a 4
- Domande delle richieste iperesponenziali di media 1 e cv pari a 4

Per ottenere i risultati sono stati utilizzati sempre 128 batch di dimensione 512.

I risultati dello studio del gruppo 1 sono visibili eseguendo la simulazione dopo aver impostato nel file di `config.properties`

```
experiment=adv1
```

mentre quelli del gruppo due impostando

```
experiment=adv2
```

16.1 Gruppo 1: soluzione metodo di autoscaling

In questi esperimenti si vuole scoprire quale dei due metodi di autoscaling è il migliore ("r0" è quello che monitora il tempo di risposta, "jobs" invece monitora la popolazione nel sistema), in modo da fissare questa variabile negli esperimenti successivi.

Negli esperimenti sono stati fissati i parametri di SI^{max}, Max num. WS e Start num. WS impostati alle costanti 2, 10 e 5.

I risultati sono riportati in figura 17 e in tabella 23, dove per brevità sono stati riportati solo quelli che hanno il minore numero di violazioni per entrambi i metodi.

Da porre attenzione che l'unità di misura del parametro "Soglia scaling" dipende dal tipo di scaling (se è "r0" saranno secondi, se è "jobs" sarà il numero di job per server attivo). Proprio per questo motivo, per meglio visualizzare i risultati di questo studio, in figura 17 sono state riportate le violazioni allo SLO non in funzione della soglia di scaling ma della capacità allocata al secondo per evidenziare quale indicatore, a parità di violazioni, alloca meno capacità.
Quello che si può vedere da questo grafico è che il metodo che usa il numero di job è quello che riesce a rispettare meglio gli SLO al 95%, al tempo stesso allocando meno capacità, motivo per cui si decide di fissarlo come metodo di scaling nei successivi esperimenti.

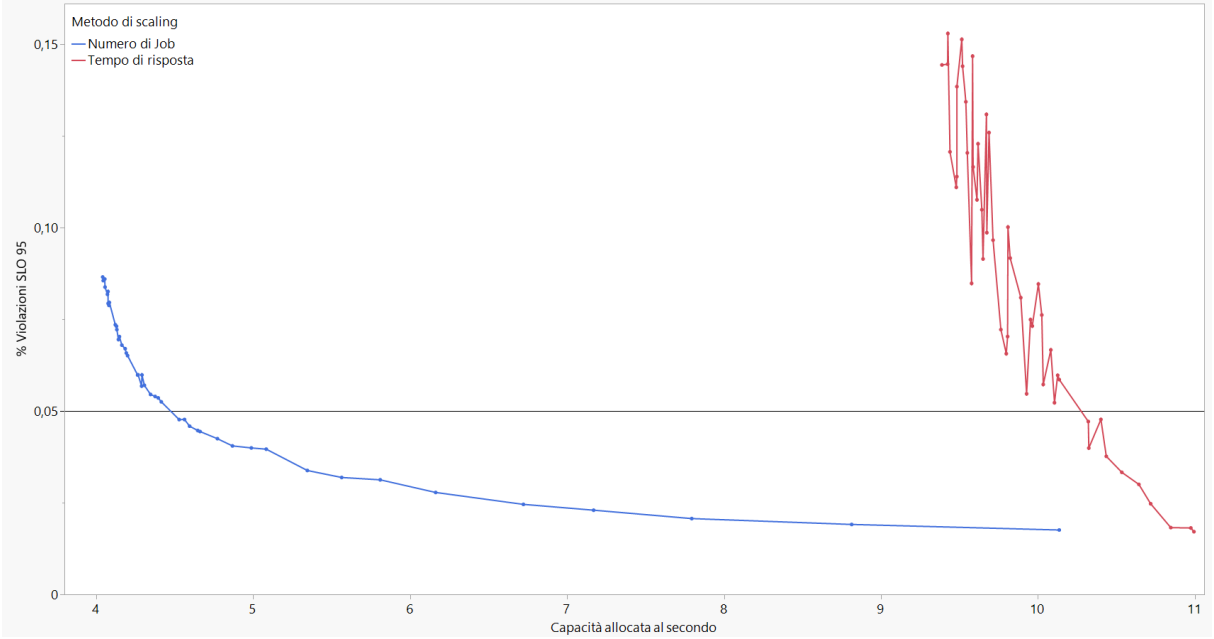

Figura 17: Esperimenti gruppo 1

Tabella 23: Esperimenti gruppo 1

Esp.	Soglia scaling	Tipo scaling	% Violazioni SLO	Cap. al sec.
101	0.2 s	r0	0.0171206226 ± 0.0016617587	10.996093019 ± 0.0064418485
102	0.4 s	r0	0.0171206226 ± 0.0016617587	10.994890557 ± 0.008424475
103	0.6 s	r0	0.0171206226 ± 0.0016617587	10.994607851 ± 0.0088906011
104	0.8 s	r0	0.018127718 ± 0.0021644094	10.974860931 ± 0.0204636098
105	1.0 s	r0	0.0182345312 ± 0.0018712951	10.847112327 ± 0.0730943245
106	1.2 s	r0	0.0247348745 ± 0.0045510291	10.718768015 ± 0.1287418384
107	1.4 s	r0	0.0300144961 ± 0.0055307062	10.643773599 ± 0.1210049017
201	0.1	jobs	0.0175783932 ± 0.0016464862	10.136176994 ± 0.0773851974
202	0.2	jobs	0.0190890364 ± 0.0016235611	8.8134800038 ± 0.1423817371
203	0.3	jobs	0.0206912337 ± 0.0017032438	7.7940051853 ± 0.1838093656
204	0.4	jobs	0.022980087 ± 0.0015963248	7.1688169337 ± 0.1995667167
205	0.5	jobs	0.0245670253 ± 0.0015014792	6.7213835524 ± 0.2084280528
206	0.6	jobs	0.0278171969 ± 0.0015360637	6.1623113299 ± 0.219926815
207	0.7	jobs	0.0312657359 ± 0.0019451192	5.8089271367 ± 0.2253092598

16.2 Gruppo 2: ricerca configurazione migliore

Gli esperimenti effettuati sono riportati in tabella 24, per brevità sono riportate solo le configurazioni con Max num. server pari a 10.
I risultati sono visibili in figura 18 e 19: sulle ascisse troviamo SI^{max}, raggrupati per la Soglia di scaling (visibile in alto), mentre sulle ordinate si mostrano in figura 18 le percentuali di Violazione degli SLO 95, e in figura 19 la Capacità allocata al secondo.
Al diminuire della capacità allocata e del numero di server disponibili aumentano i casi di violazione degli SLO, ma in questi esperimenti troviamo casi in cui gli SLO sono rispettati allocando una capacità al secondo molto prossima a quella teorica calcolata. Si nota inoltre come siano preferibili soglie di SI^{max} molto basse, in accordo con i risultati del caso base.

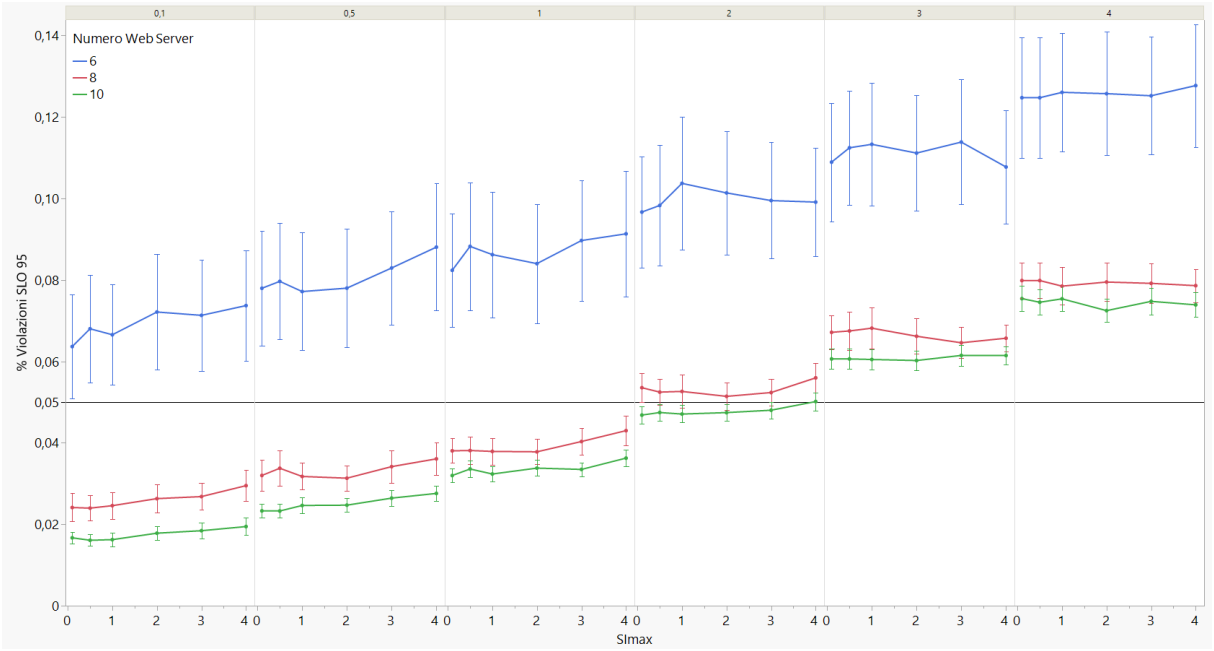

Figura 18: Esperimenti gruppo 2 sulla Percentuale di violazioni degli SLO 95

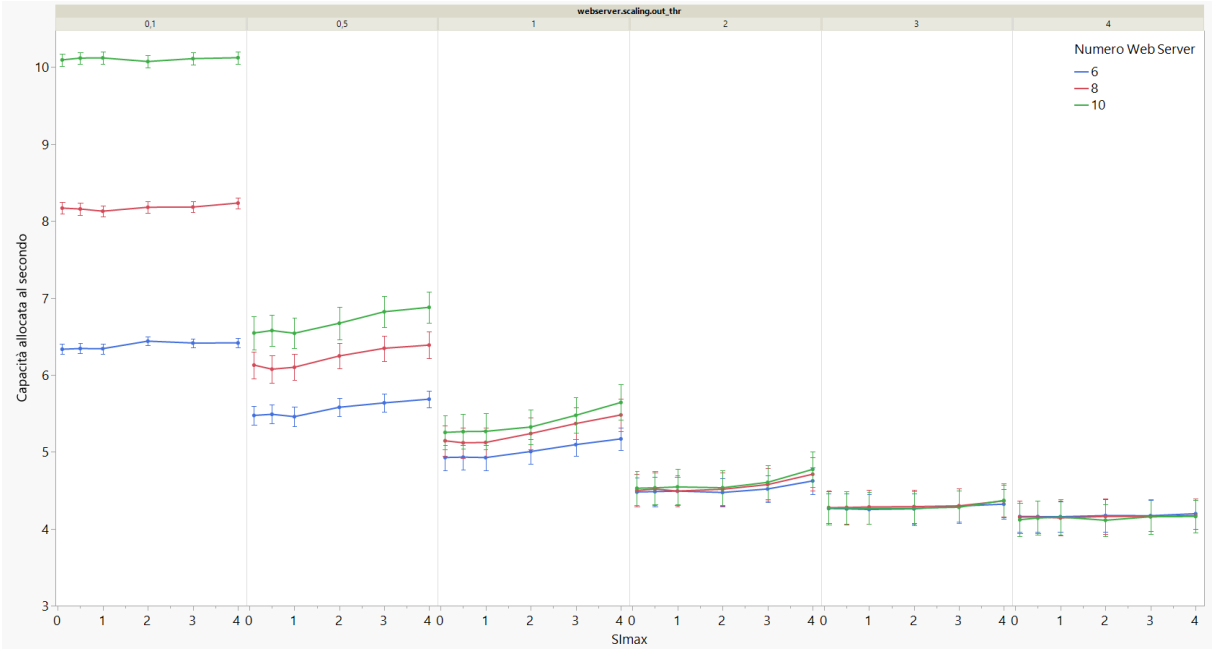

Figura 19: Esperimenti gruppo 2 sulla Capacità allocata al secondo

Tabella 24: Esperimenti gruppo 2

Esp.	SI^{max}	Soglia scaling	% Violazioni SLO	Cap. al sec.
27	0,1	0,5	0,0233005264 ± 0,0016599551	6,5450209001 ± 0,2127878051
28	0,1	1	0,0319829099 ± 0,0017477613	5,2529229238 ± 0,2235987083
29	0,1	2	0,0468146792 ± 0,0021080925	4,52705842 ± 0,2171570477
57	0,5	0,5	0,0232700084 ± 0,0016537535	6,5775243661 ± 0,2024260967
58	0,5	1	0,0335851072 ± 0,0019771719	5,263040621 ± 0,2265701753
59	0,5	2	0,0474555581 ± 0,0020269165	4,5325345322 ± 0,2165193519
87	1	0,5	0,0246280613 ± 0,0019848014	6,5417575064 ± 0,1969516187
88	1	1	0,0323491264 ± 0,0018983909	5,2663214455 ± 0,231218127
89	1	2	0,0470740826 ± 0,0021254314	4,5446528451 ± 0,2303234247
117	2	0,5	0,0247043565 ± 0,0016881744	6,6720081651 ± 0,2144586183
118	2	1	0,0337987335 ± 0,0019398952	5,3227710975 ± 0,2249713005
119	2	2	0,0474250401 ± 0,0020223107	4,5337995373 ± 0,2247230623
147	3	0,5	0,0264286259 ± 0,0019500329	6,8212770089 ± 0,2057095806
148	3	1	0,033478294 ± 0,0017098519	5,4749585424 ± 0,2282977435
149	3	2	0,0480201419 ± 0,0020450784	4,6033740781 ± 0,2219238174
177	4	0,5	0,0275883116 ± 0,0019071455	6,8788875055 ± 0,2056770331
178	4	1	0,036255436 ± 0,0020334666	5,6422352375 ± 0,23049469
179	4	2	0,050141146 ± 0,0022228381	4,7725138236 ± 0,2292276199

17 Conclusioni

17.1 Meccanismo di routing dinamico

Il meccanismo di routing in caso di congestione suggerito dal libro presenta dei gravi problemi di instabilità scegliendo soglie basse.
Questa instabilità è dovuta a situazioni in cui lo Spike Server stesso diventa congestionato e non riesce a completare i job, diventando il collo di bottiglia del sistema. In questa situazione l'indicatore del numero di job SI non riesce a scendere sotto la soglia $SI^{\max}$, per cui i nuovi arrivi vengono sempre instradati verso lo Spike Server nonostante il suo già elevato stato di congestione e le violazioni agli SLO aumentano vertiginosamente.
Al contrario, la soluzione proposta risolve questo problema tenendo in considerazione un'eventuale stato di congestione anche dello Spike Server, e questo migliora le performance in generale come osservato in figura 7.

17.2 Necessità di sovradimensionamento

La presenza di fluttuazioni long-term nel caso del sistema base senza autoscaling comporta la necessità di sovradimensionare il sistema per non provocare troppe violazioni dello SLO durante il periodo di picco degli arrivi. Questa misura provoca però un utilizzo inefficiente delle risorse durante i periodi a bassa intensità di carico.
Tale problematica ha suggerito di utilizzare la tecnica di autoscaling come miglioramento del sistema: l'autoscaler risulta molto efficiente nello scalare il sistema in base alla richiesta, e questo permette di adattare l'allocazione delle risorse in funzione del carico in arrivo.

17.3 Indicatore per l'autoscaling

L'indicatore di scaling scoperto essere il migliore è stato quello basato sul numero di job nel sistema: questo è spiegabile dal momento che tale metrica può essere aggiornata con frequenza più regolare rispetto a quella del tempo di risposta originariamente proposta.
In particolare, utilizzare il tempo di risposta è problematico dal momento che per avere un nuovo campione di questa metrica è necessario un completamento, ma in periodi di congestione questi avvengono molto più di rado a causa della politica processor sharing adottata, di conseguenza la finestra scorrevole si aggiorna molto meno e le azioni di scaling sono più rare. Al contrario, la metrica della popolazione può essere campionata anche in corrispondenza degli arrivi, che non subiscono rallentamenti come i completamenti, e questo aumenta la responsiveness dell'autoscaler ai picchi di carico.

17.4 Effetto dello Spike Server

Per l'obiettivo del "right sizing problem" risulta essere migliore utilizzare sempre soglie di $SI^{\max}$ molto basse. Questo suggerisce che, a parità di capacità, è sempre meglio avere un Web Server in più rispetto che uno Spike Server adibito specificatamente per la gestione delle congestioni.

17.5 Confronto dei risultati

In conclusione, si confrontano la "migliore" configurazione trovata del sistema base e la migliore configurazione trovata del sistema migliorato: per la prima prendiamo l'esperimento 219 del gruppo 2, mentre per la seconda l'esperimento 29 del gruppo 2.
Per "migliore" si intende la configurazione la cui percentuale delle violazioni dello SLO è inferiore a 5% per tutto l'intervallo di confidenza calcolato, e che utilizza la capacità allocata al secondo minore.

Esperimento	SI^{max}	Soglia scaling	Num. WS	% Viol. SLO	Cap. al sec
base-group2-219	0,4	/	7	0,035309 ± 0,006651	8,0 ± 0,0
adv-group2-29	0,1	2	10	0,046814 ± 0,002108	4,52705 ± 0,217157

Si può notare come il sistema migliorato permette di rispettare lo SLO utilizzando solo il 56.6% della capacità allocata dal sistema base.

18 Appendice

18.1 Trasformazione di Box-Muller

Siano U_1 e U_2 due variabili aleatorie indipendenti ed uniformemente distribuite nell'intervallo $(0, 1]$. Sia:

$$Z_0 = Rcos(\theta) = \sqrt{-2\log U_1}\cos(2\pi U_2)$$

e

$$Z_1 = Rsin(\theta) = \sqrt{-2\log U_1}\sin(2\pi U_2)$$

Allora Z_0 e Z_1 sono variabili aleatorie indipendenti con distribuzione normale di devazione standard unitaria.

18.2 I test delle variabili aleatorie

Per testare che effettivamente le variabili aleatorie implementate si comportassero come le distribuzioni teoriche sono stati costruiti degli intervalli di confidenza per la media campionaria utilizzando la diseguaglianza di Chebyshev:

$$P(|X - \mu_X| > \epsilon) \geq \frac{\sigma_X^2}{\epsilon^2}$$

Nel nostro caso X corrisponde alla media campionaria X_n per la quale sappiamo che:

$$E[X_n] = \mu_X$$

e

$$D^2[X_n] = \frac{\sigma_X^2}{n}$$

Dove X è al variabile aleatoria, che segue una certe distribuzione, dalla quale effettuiamo gli n campionamenti per calcolare poi la media campionaria. Dunque possiamo scrivere la diseguaglianza come segue:

$$\begin{aligned}
P(|X_n - \mu_X| > \epsilon) &\leq \frac{\sigma_X^2}{n\epsilon^2} \\
P(|X_n - \mu_X| \leq \epsilon) &\geq 1 - \frac{\sigma_X^2}{n\epsilon^2} \\
P(-\epsilon \leq X_n - \mu_X \leq \epsilon) &\geq 1 - \frac{\sigma_X^2}{n\epsilon^2} \\
P(X_n - \epsilon \leq \mu_X \leq X_n + \epsilon) &\geq 1 - \frac{\sigma_X^2}{n\epsilon^2}
\end{aligned}$$

Dunque abbiamo costruito un intervallo di confidenza per la media di una variabile aleatoria generica. Ovvero, con una probabilità $\geq 1 - \frac{\sigma_X^2}{n\epsilon^2}$ possiamo dire che il vero valore della media cade nell'intervallo $[X_n - \epsilon, X_n + \epsilon]$. Per fare il test vengono fissati il numero di ripetizioni n e la confidenza α. Dalla definizione di intervallo di confidenza fissiamo:

$$1 - \alpha = 1 - \frac{\sigma_X^2}{n\epsilon^2} \Rightarrow \epsilon = \sqrt{\frac{\mu_X^2 \cdot C^2}{n \cdot \alpha}}$$

In questo modo per ciascun n e α passati come parametri al test viene calcolato ϵ che ci permette di dire con probabilità $1 - \alpha$ che il vero valore della media della variabile aleatoria ricade nell'intervallo di confidenza.
Per quanto riguarda il vero valore della varianza dell'iperesponenziale invece si è utilizzata sempre la diseguaglianza di Chebyshev osservano che:

$$\mu_X = \sqrt{\frac{\sigma_X^2}{C^2}}$$

Dunque possiamo riscrivere la diseguaglianza come:

$$P\left(C^2 \cdot (X_n - \epsilon^2) \leq \sigma_X^2 \ \leq C^2 \cdot (X_n + \epsilon^2)\right) \geq 1 - \frac{\sigma_X^2}{n\epsilon^2}$$

Analogamente al caso della media si calcola ϵ in funzione di α e n e abbiamo costruito un intervallo di confidenza per la varianza dell'iper-esponenziale che ci permette di dire che con probabilità $1-\alpha$ il vero valore della varianza cade all'interno dell'intervallo.
Per la distribuzione normale invece è stato utilizzato l'intervallo di confidenza notevole per la varianza:

$$\left(\frac{(n-1) \cdot S_n^2}{\chi^{2,+}_{n-1,\alpha/2}}, \frac{(n-1) \cdot S_n^2}{\chi^{2,-}_{n-1,\alpha/2}} \right)$$

dove:

- S_n^2 è la varianza campionaria di una distribuzione normale. Vengono fatti n campionamenti.
- $\chi^{2,+}_{n-1,\alpha/2}$ è il valore critico della coda superiore di livello $\frac{\alpha}{2}$ di una distribuzione χ^2 con $n-1$ gradi di libertà.
- $\chi^{2,-}_{n-1,\alpha/2}$ è il valore critico della coda inferiore di livello $\frac{\alpha}{2}$ di una distribuzione χ^2 con $n-1$ gradi di libertà.

Sia X una variabile aleatoria, allora per definizione il valore critico della coda inferiore di livello α di X è il minimo α-quantile. Rispettivamente il valore critico della coda superiore di livello α di X è il massimo α-quantile.