ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea in Ingegneria Informatica Magistrale

Progetto Opzionale

di

Sistemi Distribuiti M

Load Balancing e Spring Cloud: esempi di utilizzo e test di performance

Docente: Paolo Bellavista

Corso: Sistemi Distribuiti

Studente: Paolo Verdini

Matricola: 0000946756

Anno Accademico: 2020/2021

Indice

Introduzione

Nei sistemi distribuiti, il monitoraggio è un'operazione di fondamentale importanza, in quanto dà modo di capire come gestire tutte le risorse utilizzate da un'applicazione. Monitorare un'applicazione distribuita significa ricevere informazioni dicarico del sistema in ogni momento, per poterlo gestire come meglio vogliamo, utilizzando eventi, statistica e dati storici oppure osservando a intervalli limitati. Le informazioni raccolte vengono utilizzate per prevedere possibili prossimi utilizzi di risorse. Dal momento che non si ricevono tutte le informazioni in tempo reale, ma solo una porzione di essa (snapshot), se non si ha una situazione di continuità di monitoraggio, potremmo avere uno scenario reale estremamente diverso da quello rappresentato dai dati, che non sono freschi.

È evidente la necessità di limitare il costo della raccolta delle informazioni e manutenzione per limitare l'intrusione. Nei sistemi generici il monitoraggio non dispone di risorse dedicate, ma deve usare quelle che sono sfruttate anche dall'applicazione. Questa concorrenza suggerisce di limitare al massimo gli impegni di tali risorse in modo da limitare la percentuale di esse portate via all'applicazione: più usiamo le risorse dell'infrastruttura per monitorare e scambiare dati con applicazione, più limitiamo quelle dedicate all'applicazione. L'obiettivo delle infrastrutture di uso generale è quello di avere la minima percentuale possibile di risorse dedicate al monitoraggio. I dati disponibili dall'applicazione sono in generale una quantità enorme, ma è necessario sceglierne solo un insieme riepilogativo per la raccolta e monitoraggio: probabilmente solo una certa occupazione di banda media in un periodo limitato; oppure l'applicazione invia delle informazioni in risposta a degli specifici eventi, limitando così lo spreco di risorse.

Quanto detto finora si riassume nel principio di "minimal intrusion": qualsiasi funzione di supporto all'applicazione (diciamo alla vera logica di business) deve limitare il costo di funzionamento al minimo, compatibilmente con il raggiungimento del suo obiettivo, in modo da intromettersi al minimo nel sistema.

La decisione di come organizzare i componenti in partizioni e in server diversi dipende dalla strategia di allocazione che si segue. Esistono due principali tipi di allocazione:

- Allocazione statica: le risorse da utilizzare in un certo server sono decise a priori e una volta allocate non possono essere spostate.

- Allocazione dinamica: le risorse da utilizzare in un certo server sono decise runtime.

Se utilizziamo la tipologia statica, è necessario identificare staticamente in che nodi allocare le risorse, senza poterle poi spostare: in questo modo viene effettuato un cosiddetto "load sharing" tra i nodi durante l'esecuzione. Se utilizziamo, invece, la tipologia dinamica, è possibile allocare le risorse in un nodo e successivamente migrarle in altri nodi, in modo da ottenere un'efficienza globale. In questo caso parliamo di "load balancing".

Nel caso di load sharing, la decisione di allocazione di risorse è presa "out of band", prima dell'esecuzione dell'applicazione. Questo dà modo di ottimizzare gli algoritmi di predizione del carico. Lo svantaggio più evidente di questa strategia è l'inflessibilità runtime.

Nel caso di load balancing, abbiamo un modello di allocazione che impatta runtime, togliendo risorse computazionali all'applicazione. Il vantaggio più grande è la flessibilità e l'allocazione delle risorse in modo dinamico, così da ottimizzare il carico in ogni situazione.

Load Balancing

Nonostante i costi più elevati rispetto al load sharing, il load balancing è la strategia più utilizzata nei sistemi distribuiti reali, in quanto offre la possibilità di gestire efficientemente le risorse in differenti server. Le soluzioni per il load balancing sono divise in due modelli, uno più statico e uno più dinamico.

Nell'approccio statico, è assunta la conoscenza a priori dello stato globale del sistema distribuito, in modo tale da predire il requisito di risorse di lavoro e, per esempio, il tempo di comunicazione necessario. In questo metodo, il load balancing è raggiunto tramite un mapping o un assegnamento da un set di operazioni a un set di processori, in modo tale da minimizzare una cosiddetta funzione di performance. Al tempo stesso, questo approccio può prevedere sia un assegnamento deterministico che probabilistico. Nel caso deterministico, si ha che il nodo i delega degli "extra task" al nodo j in ogni evenienza in cui è necessario; nel caso probabilistico, il nodo i delega gli extra task al nodo j con probabilità p, al nodo k con probabilità q. Il principale svantaggio dell'approccio statico è che, quando vengono prese le decisioni di load balancing, non si prende in considerazione lo stato corrente del sistema. Questo ha un impatto

molto importante sulle prestazioni del sistema stesso, a causa dell'imprevedibilità di flusso del carico nel sistema distribuito.

Nell'approccio dinamico, il load balancing è innescato a seguito di un'analisi dello stato corrente del sistema: i task sono liberi di muoversi dinamicamente da un nodo "overloaded" verso uno "underloaded", in modo tale da garantire un servizio più efficiente. Questa capacità di reagire ai cambiamenti del sistema è il principale vantaggio dell'approccio dinamico del load balancing. Infine, trovare una soluzione dinamica è molto più complicato di una soluzione statica, anche se l'approccio dinamico dà modo di ottenere una prestazione ottimale, proprio per il fatto di poter analizzare runtime il carico del sistema.

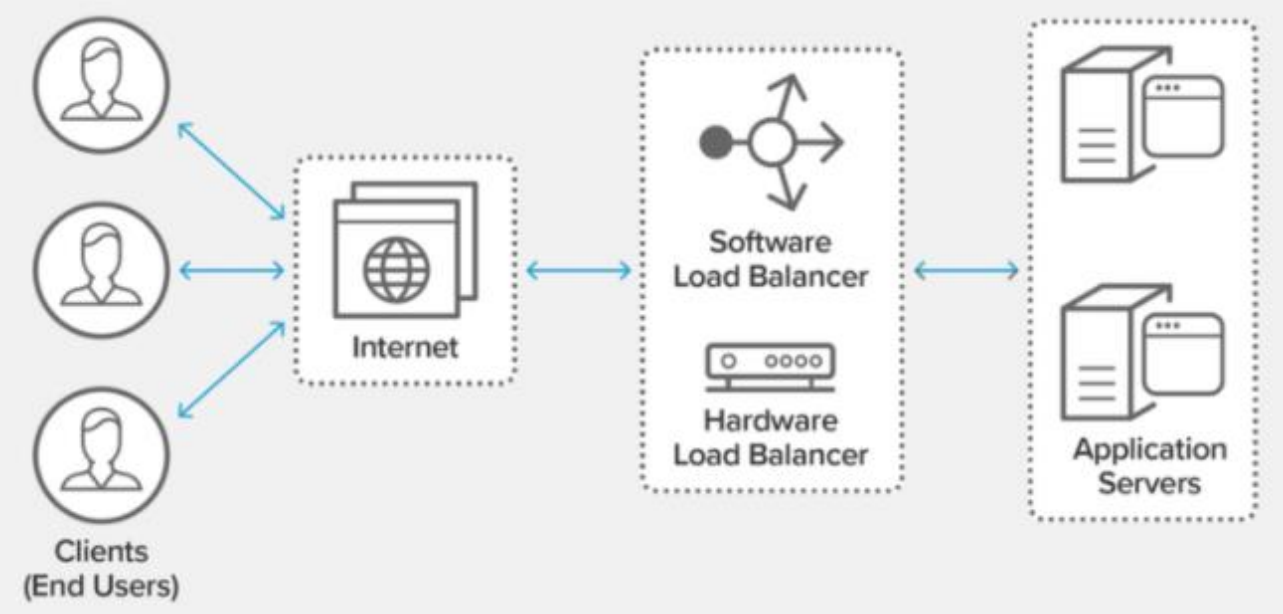
Andando nel concreto, il load balancing può essere strutturato come un'efficiente ridistribuzione delle richieste che stanno arrivando dalla rete verso un gruppo di server che risiedono nel backend del sistema distribuito (server pool). Per esempio, le applicazioni web moderne devono servire centinaia di migliaia, se non milioni, di richieste concorrenti da parte di utenti o clienti e devono processare immagini, testi, video velocemente e in modo affidabile. Per riuscire a garantire questi requisiti, è generalmente necessario aggiungere diversi nodi server al sistema distribuito, che devono però essere gestiti al meglio, tramite le tecniche sopra citate di load balancing (principalmente con approccio dinamico).

Load Balancer

Un Load Balancer può essere visto come un "vigile urbano" del traffico della rete, che risiede davanti ai server e fa il routing delle richieste dei clienti verso tutti i server capaci di soddisfarle. Il compito principale è di garantire velocità e utilizzo della capacità dei server stessi in maniera ottimale, in modo tale che nessun server sia overloaded, cosa che potrebbe degradare le prestazioni. Se uno specifico server va down, il load balancer redirige il traffico ai restanti server che sono in esecuzione. Quando un nuovo server viene aggiunto al server pool, il load balancer inizia automaticamente ad inviargli richieste.

Il load balancer possiede quindi alcune funzionalità:

- Distribuzione delle richieste dei clienti o del carico di rete in modo efficiente su più server
- Elevata disponibilità e affidabilità, inviando richieste solo ai server online
- Flessibilità di aggiungere o eliminare server dal server pool, in base a quanto richiesto e al carico del sistema corrente



Algoritmi di Load Balancing

Esistono differenti algoritmi di load balancing, ciascuno con i propri benefici; la scelta del metodo di load balancing dipende dai bisogni del sistema distribuito in questione e dal contesto di utilizzo.

- Round Robin: le richieste sono distribuite sequenzialmente ai server del server pool.
- Least Connections: una nuova richiesta è indirizzata verso il server con il minor numero di connessioni a clienti corrente.
- Least Time: le richieste sono indirizzate a un server tramite un algoritmo che combina due parametri, ovvero velocità di risposta e poche connessioni attive.
- Hash: le richieste sono distribuite tramite una chiave definita dallo sviluppatore, per esempio indirizzo IP, URL della richiesta ecc.

- IP Hash: l'indirizzo IP del cliente è il parametro che discrimina a quale server associare la richiesta.
- Random Two Choices: selezioniamo due server in maniera casuale e mandiamo la richiesta a quello che tra i due è selezionato applicando una delle tecniche sopra elencate (generalmente Least Connections).

Microservizi

I microservizi sono un approccio per sviluppare un'architettura dei software secondo cui questi ultimi sono composti di servizi indipendenti di piccole dimensioni, che comunicano tra di loro mediante delle API ben definite. Le architetture dei microservizi permettono di scalare e sviluppare applicazioni in modo più rapido e veloce, dando modo di promuovere innovazione e accelerare il time-to-market di funzionalità aggiuntive.

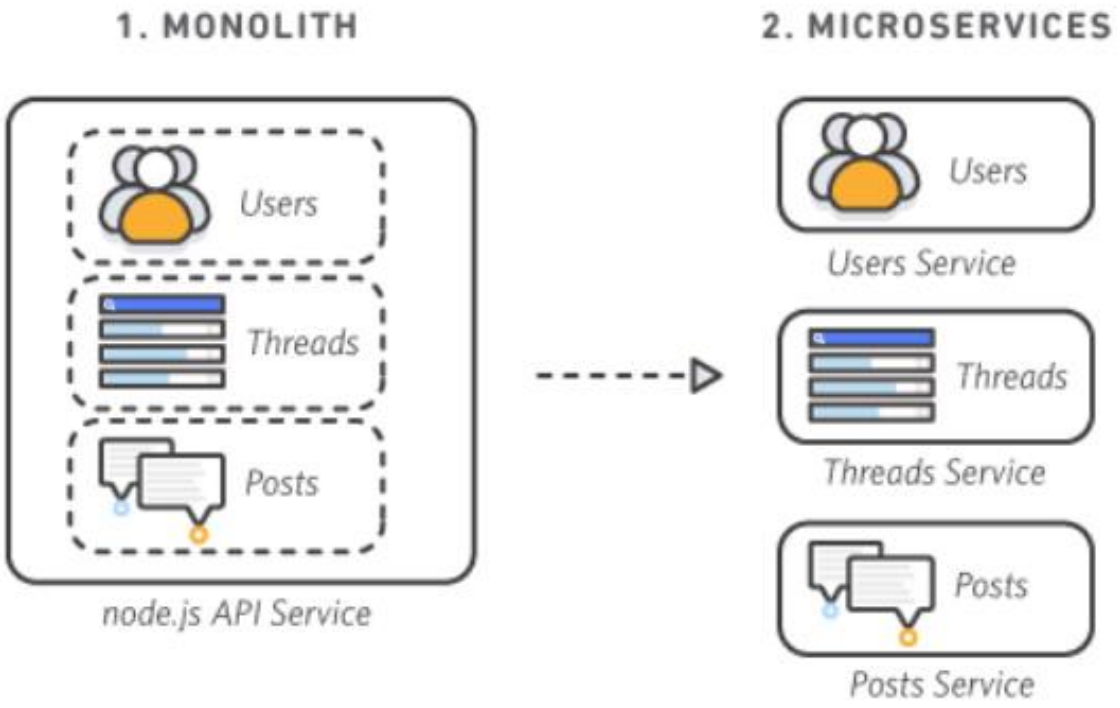
In un'architettura monolitica, tutti i processi sono collegati tra di loro in modo stretto e vengono eseguiti come un unico servizio: questo significa che se un solo processo del servizio ha un picco di richieste, è necessario ridimensionare l'intera architettura. Migliorare o aggiungere funzionalità in un'architettura monolitica diventa sempre più complesso all'aumentare del volume del servizio stesso; questa complessità limita l'innovazione e l'upgrade con nuove idee del sistema stesso. Infine, le applicazioni monolitiche rappresentano un punto di rischio molto ampio, in quanto, essendo tutti i processi sono collegati, aumenta l'impatto di errore di un singolo processo sull'intera applicazione.

Un'applicazione ad architettura basata su microservizi ha una realizzazione a componenti indipendenti, che eseguono ciascun processo applicativo come un servizio. Come detto precedentemente, tali servizi comunicano con API ben definiti e leggere, in modo tale da rendere scalabile e flessibile l'applicazione. Poiché ogni servizio è eseguito in modo indipendente, questo può essere aggiornato, distribuito e ottimizzato per rispondere alla richiesta di nuove funzionalità o funzioni specifiche di un'applicazione senza impattare sugli altri servizi.

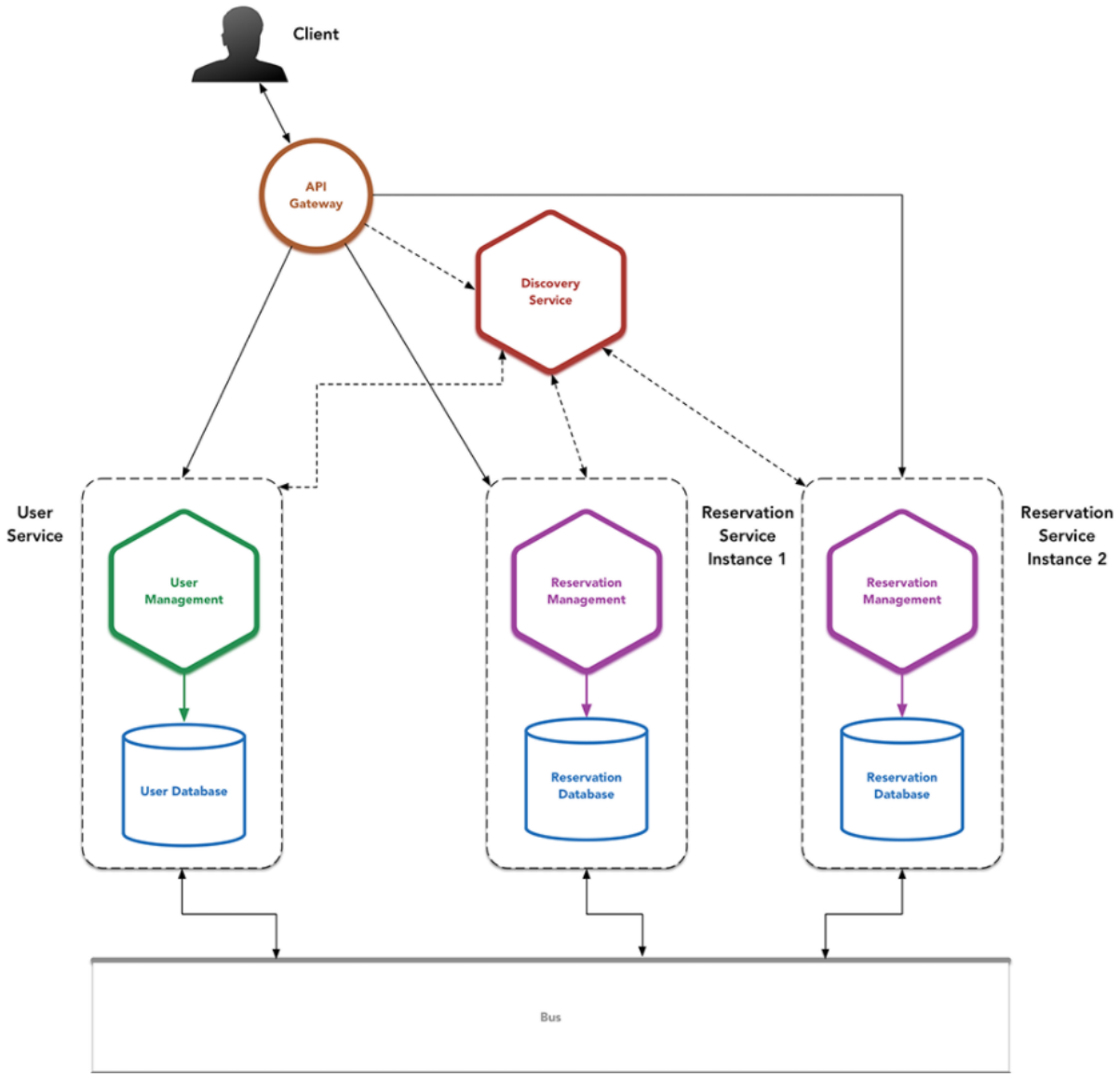
I servizi non condividono codice o implementazione con gli altri servizi, grazie all'accesso tramite API. Ogni servizio è progettato per una particolare funzionalità e si concentra a

svolgerla al meglio, in maniera ottimizzata: se gli sviluppatori aggiungono funzionalità specifiche a un servizio, rendendolo più complesso, esso può a sua volta essere scomposto in altri microservizi.

Scalabilità, integrazione, distribuzione continua e flessibilità sono i punti di forza dei microservizi: possibilità di integrare nuove funzionalità in modo accurato e semplice, errori che influiscono meno sul costo delle operazioni, possibilità di lavorare in parallelo con team diversi, ciascuno specializzato in un particolare microservizio.



Lo scenario, passando da un'architettura monolitica a una a microservizi, è quello mostrato in figura.



È evidente che nascono problematiche per la gestione dei microservizi, le quali vanno risolte in modo efficiente ed ottimizzato:

- Ogni servizio deve sapere dove sono gli altri: c'è, quindi, bisogno di un "Service Registry" che permetta la "Service Discovery", cioè un registro che permette ai servizi di registrarsi e di trovarsi a vicenda.
- Le configurazioni di sistema devono essere distribuite in tutti i moduli: i sistemi di "Configuration Management" permettono di centralizzarle e distribuirle.
- Un sistema distribuito deve essere tollerante al fallimento di una delle sue parti, quindi se uno o più moduli non risultano disponibili, il sistema deve essere in grado di isolare quei moduli e dare una risposta di fallback (tecnica chiamata "Circuit Breaker").

- L’entrypoint del sistema deve rimanere sempre uno, in modo da non far percepire la suddivisione nei servizi sottostanti (API Gateway).
- In un sistema distribuito è necessario accettare la concezione di “eventually consistent”, piuttosto che una transazionalità di tipo ACID, che sarebbe troppo difficile da mantenere.
- Le dipendenze dei moduli non sono più a tempo di compilazione, ma risiedono in un’interfaccia comune con la quale si scambiano i dati. Se non si fa separazione tra responsabilità di ogni servizio, si rischia di entrare in referenze cicliche.

Microservizi in Java

Come ben sappiamo, per riuscire a creare un’applicazione distribuita si è sempre utilizzato un Application Server (tipo JBoss) oppure un Servlet Container (tipo Tomcat), in cui fare il deploy del nostro artefatto, costituendo un unico ambiente dove girano anche altre applicazioni. Quando parliamo di microservizi, invece, è necessario ribaltare il punto di vista: l’artefatto stesso esegue un server che ha un ciclo di vita pari a quello dell’applicazione. Con la diffusione dell’architettura REST e la necessità di rendere scalabili e flessibili i servizi, utilizzando metodologie stateless, l’architettura a microservizi è riuscita ad esplodere in maniera netta. Ma per potersi concretizzare, è necessario avere strumenti moderni che supportino le ben conosciute problematiche dei sistemi distribuiti.

La questione che ha portato nel tempo a far nascere una delle soluzioni a tutte queste problematiche, in ambiente Java, è proprio rispondere a come riuscire a sviluppare i concetti sopra elencati dei microservizi, insieme agli elementi dei sistemi distribuiti.

Spring Boot

Nato nel 2012, Spring Boot è una soluzione “convention over configuration” per il framework Spring del mondo Java, il quale mira a ridurre la complessità di configurazione di nuovi progetti Spring. Spring Boot definisce una configurazione di base che include le linee guida per l’uso

del framework e tutte le librerie rilevanti di terze parti, facendo in modo di semplificare la creazione di applicazioni indipendenti e basate su Spring (proprio per questo, al giorno d'oggi molte applicazioni Spring sono basate anche su Spring Boot).

Spring Boot offre diverse funzionalità vantaggiose, che lo hanno portato ad avere molto successo:

- Incorpora direttamente server o container nell'infrastruttura, senza dover quindi gestire a mano l'uso dei file WAR o EAR.
- Grazie all'introduzione dei POM (Project Objects Models), offre una configurazione semplificata di Maven.
- Porta una configurazione automatica di Spring, dove possibile.
- Fornisce anche caratteristiche non funzionali o configurazioni esternalizzate.

Ma manca ancora un'architettura per riuscire a rispondere, in mondo Java, alle problematiche sui microservizi e sistemi distribuiti, Spring Cloud.

Spring Cloud

Spring Cloud è un insieme di soluzioni ai problemi comuni dei sistemi distribuiti, organizzati in tanti sotto-progetti: offre diversi strumenti per sviluppatore per creare, in maniera agile e scalabile, diversi pattern dei sistemi distribuiti. Il coordinamento di sistemi distribuiti conduce spesso a "boilerplate" pattern e pezzi di codice, Spring Cloud permette di mettere in piedi servizi e applicazioni che implementano tali pattern rapidamente. Lavora bene anche in SaaS.

I servizi offerti da Spring Cloud sono vasti tra le feature dei sistemi distribuiti, per esempio: routing, distributed configuration, load balancing, global clocks, leadership election, distributed messaging ecc.

Illustriamo alcuni componenti principali che fanno parte di Spring Cloud, per poi soffermarci sull'utilizzo di Spring Cloud Load Balancer, argomento del progetto in questione.

Spring Cloud Config

Spring Cloud Config offre una vasta gamma di funzionalità per la gestione delle configurazioni: si prospetta come un server centralizzato che permette di gestire configurazioni sia server-side che client-side. Di default, le configurazioni vengono scaricate da una repository git contenente una serie di file che identificano il nome dell'applicazione, il profilo attivo e una label. Spring Cloud Config permette, grazie all'utilizzo delle astrazioni "Environment" e "PropertySource", di abbracciare completamente applicazioni Spring, ma offre anche modo di essere usato anche in applicazioni che eseguono in qualunque linguaggio.

Ad esempio, quando un'applicazione si sposta dall'ambiente di sviluppo verso quello di test e alla fine verso quello di produzione, è possibile tramite Spring Cloud Config gestire le configurazioni tra questi ambienti ed essere certo che ogni applicazione abbia tutte le dipendenze e configurazioni necessarie per essere eseguite durante la migrazione.

Oltre alla configurazione di default che sfrutta le repository git, è possibile impostare l'utilizzo anche di database, file system e HashiCorp Vault.

Le caratteristiche più interessanti sono le seguenti:

- Possibilità di richiedere tramite REST la configurazione di un servizio. Immaginiamo di aver effettuato il binding di Spring Cloud Config a localhost:8080 e di avere un servizio di nome "my-service" e un profilo "profile", la configurazione di tale servizio è accessibile all'URL:

  ```
  http://localhost:8080/my-service/profile
  ```

- Le property di un servizio possono essere aggiornate dinamicamente senza la necessità di riavviarlo. È sufficiente invocare l'endpoint "/refresh" oppure "RefreshEndpoint.refresh()", anche tramite JMX. Per poter aggiornare un valore delle property iniettate, è necessario che il bean in questione sia "@RefreshScope".
- Possibilità di propagare automaticamente le modifiche alle configurazioni tramite Message Middleware, per esempio RabbitMQ.

Spring Cloud Netflix

Spring Cloud Netflix è il kernel dell'infrastruttura Spring Cloud, il quale integra Spring Boot con Netflix OSS, la piattaforma realizzata da Netflix per rispondere alle sfide dei sistemi distribuiti. Analizziamo quindi i principali componenti che Netflix OSS (e quindi Spring Cloud Netflix) mette a disposizione:

- **Eureka**: è il service registry che abilita la service discovery tra i servizi del nostro sistema. Di default, il server si avvia sulla porta 8761 e i client, se non viene configurato diversamente, possono accederci da *localhost:8761*. Viene configurato tramite il file di configurazione application.properties o application.yml.

```
1   eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
2   eureka.instance.prefer-ip-address=true
3   eureka.instance.initialInstanceInfoReplicationIntervalSeconds=5
```

- **Hystrix**: è il circuit breaker e ha la responsabilità di aprire il circuito delle chiamate fallimentari, in casi in cui una delle parti del sistema distribuito non è disponibile in un certo momento. Il suo compito è controllare se il sistema escluso è nuovamente disponibile, richiudendo il circuito.
- **Ribbon**: client side load balancer del sistema. Generalmente, in un sistema a micro-servizi, la comunicazione può essere o diretta o tramite proxy. Se la comunicazione è diretta, allora ogni servizio conosce gli altri servizi e dove sono, grazie al service registry, quindi è lato client che avviene la decisione su chi chiamare. Se la comunicazione è tramite proxy, allora ogni servizio conosce solo il proxy ed è poi lui a decidere chi chiamare. Ecco che nasce l'idea di "bilanciamento" delle chiamate alle repliche di un servizio. Se abbiamo la prima configurazione, il bilanciamento è client side, se si ha la seconda allora il bilanciamento è server side. Di default il bilanciamento che sfrutta Ribbon è il Round Robin, ma è possibile implementare nuove tecniche.
- **Feign**: client REST dichiarativo. Permette di effettuare chiamate tra servizi annotando un'interfaccia Java con annotazioni Spring MVC o JAX-RS. Oltre a dare una semplice gestione per dichiarare un client REST, Feign si appoggia a Ribbon per scegliere lato client quale servizio chiamare e sfrutta Hystrix. Usando Feign solitamente non si

gestisce a mano Ribbon o Hystrix, in quanto viene gestito automaticamente (salvo esigenze particolari).

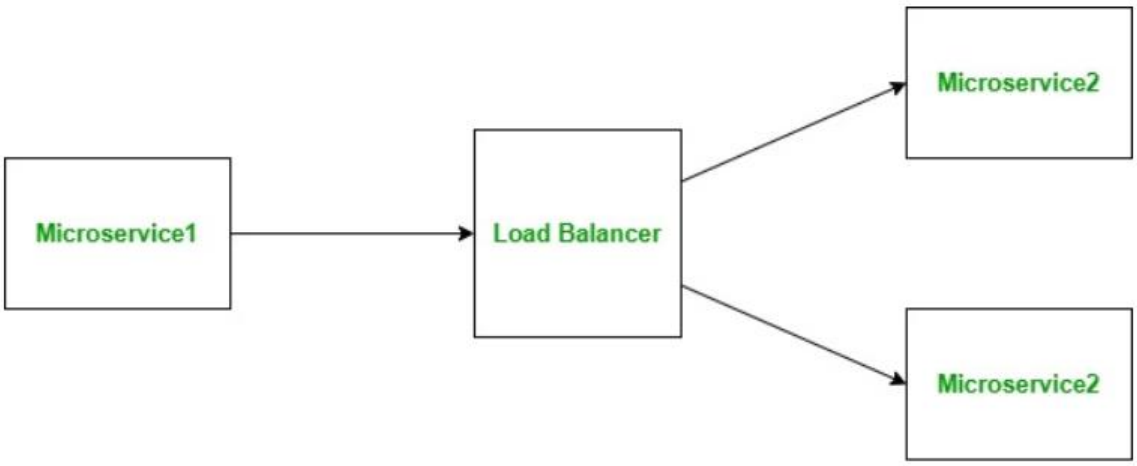
- **Zuul**: API gateway. Fa da single entry point per il sistema e redirige le chiamate ai servizi sottostanti. Si basa su Hystrix e Ribbon, in modo tale da garantire Load Balancing e Circuit Breaker. Se viene abilitato come Proxy (@EnableZuulProxy), è capace di mappare dinamicamente gli endpoint dei servizi sottostanti. Esempio, se abbiamo una risorsa */api/stand/developers* del servizio *developer-service* che è raggiungibile su una porta 8888 da localhost *localhost:8888/api/stand/developers,* allora è possibile accedere da Zuul a tale risorsa (con Zuul attivo alla porta 8080) tramite *localhost:8080/developer-service/api/stand/developer*. Zuul si rende quindi da proxy per ogni risorsa che sta sotto il servizio associato a *spring.application.name.* Una funzionalità interessante di Zuul è poter mappare una rotta nel file *zuul-gateway.yml,* in modo tale da raggiungere tramite il proxy la risorsa come se stessimo chiamando il servizio vero a proprio. Fare questa procedura permette anche di rendere omogenee le risorse all'esterno, in modo tale da nascondere la vera organizzazione nel backend. Per esempio, se abbiamo una risorsa che fa parte di un servizio di *employee-service* e che mapperebbe i developers come employees, sarebbe accessibile tramite proxy come *localhost:8080/employee-service/api/stand/developer/employees.* È però possibile, come detto sopra, creare una rotta con cui non si fa vedere all'utilizzatore la suddivisione dei servizi, per esempio creando l'entrypoint *localhost:8080/api/stand/developer/employees.* Pubblicamente l'API apparirà quindi omogenea, come ci aspetteremmo, quando in realtà nel backend è organizzata seguendo la logica del dominio.

Spring Cloud e Load Balancer

Spring Cloud offre funzionalità di Load Balancing, sia tramite Spring Cloud Load Balancer, sia tramite tecnologie che si appoggiano a Netflix OSS, con Spring Cloud Netflix. Poiché la maturità attuale di Spring Cloud Load Balancing è ancora non convincente, analizzeremo nel dettaglio la tecnologia Spring Cloud Netflix e come si possa costruire un load balancer. Per

poter illustrarne il funzionamento, è necessario specificare che esistono due tipologie di Load Balancing: server-side e client-side.

Server Side Load Balancing



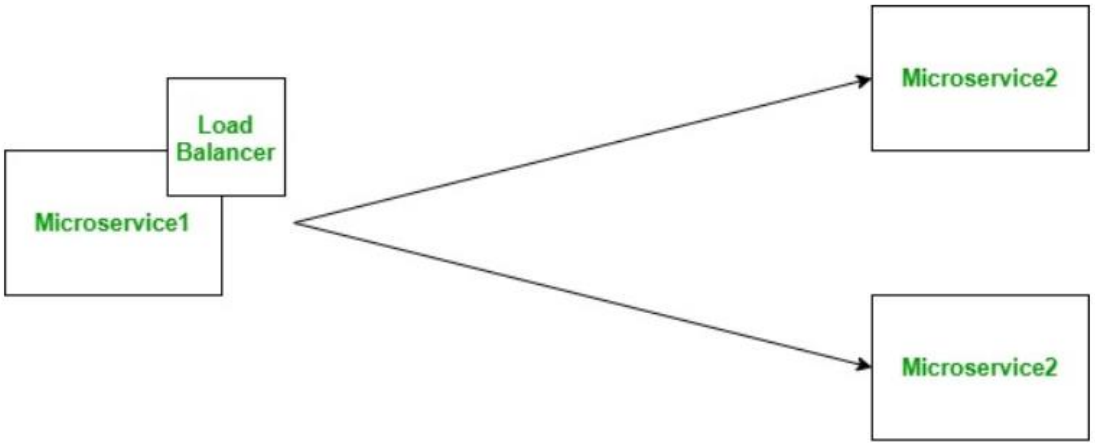
Server side load balancing

Il server-side Load Balancing consiste in una molteplicità di istanze del servizio di cui è stato fatto il deploy in multipli server e di un Load Balancer messo davanti a loro. Solitamente è un hardware Load Balancer, ma potrebbe essere anche software. Questo Load Balancer decide quindi, tramite un algoritmo predefinito, a quale server redirigere una particolare richiesta.

I principali svantaggi della modalità server-side sono i seguenti:

1. Il Load Balancer agisce come un "single point of failure", in quanto se questo cade, ogni istanza del microservizio diventa inaccessibile perché solo il Load Balancer ci fa accedere ad esse.
2. Poiché ogni microservizio avrà distinti Load Balancer, la complessità generale del sistema aumenta e può diventare difficile da gestire.
3. La latenza di rete aumenta perché il numero di hop che deve fare una richiesta passa da uno a due: uno per il Load Balancer, un altro da Load Balancer al microservizio.

Client Side Load Balancing



Client side load balancing

Nel client-side Load Balancing, le istanze del servizio sono comunque messe in esecuzione in server multipli, ma la logica del Load Balancer è parte del cliente stesso: mantiene una lista di server e decide, tramite determinati algoritmi, a quale server redirigere una particolare richiesta. I client side Load Balancer sono spesso chiamati software Load Balancer.

Il principale svantaggio del client-side Load Balancer è il fatto che la logica del bilanciamento viene mischiata alla logica del microservizio.

Ribbon

Ribbon è un Inter Process Communication Library con un built-in client side Load Balancer e fa parte del Netflix OSS. Le principali funzionalità di Ribbon sono:

- **Load Balancing**: prevede una gestione della metodologia client side Load Balancing.
- **Fault Tolerance**: può essere usato per determinare se un server è correttamente in esecuzione oppure no, ignorando quelli bloccati, in modo da non mandargli richieste.
- **Regole di Load Balancing configurabili**: Di default Ribbon utilizza Round Robin come algoritmo di Load Balancing, ma è possibile utilizzare Availability Filtering Rule e Weighted Response Time Rule. La Availability Filtering Rule sfrutta un meccanismo

in cui si analizza un server e quelli che risultano “circuit tripped” o con alto tasso di connessioni concorrenti sono ignorati e non utilizzati per assegnare la richiesta corrente. Per “circuit tripped” si intende un server il quale ha fallito per tre volte la connessione con un RestClient. Lo stato di circuit tripped permane per 30 secondi prima che il circuito venga nuovamente chiuso. Se diventa nuovamente circuit tripped, il tempo di permanenza in questo stato aumenta esponenzialmente a seguito dei fallimenti consecutivi. La Weighted Response Time Rule risponde ad una logica tramite la quale a ciascun server è associato un peso, relativo al loro tempo di risposta medio. Più è alto il tempo di risposta, minore è il peso che il server avrà associato. Questa regola seleziona casualmente un server la cui probabilità di essere selezionato è determinata dal peso associato.

- **Supporta molti protocolli**, tipo HTTP, TCP, UDP.

Configurazione e funzionamento

Per poter sfruttare le funzionalità di Ribbon all’interno di progetti Spring Cloud è necessario inserire la dipendenza alle API dentro il file *pom.xml*.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

Una volta impostata la dipendenza, si procede con la configurazione della main application, che dovrà essere una classe marcata con l’annotazione “@SpringBootApplication”. Per attivare la funzionalità di Load Balancing, è necessario che tale classe abbia anche un altro tipo di annotazione, “@RibbonClient”. Quest’ultima viene spesso utilizzata con due parametri: uno è il nome del client, uno è la classe di configurazione del Load Balancer.

```
@RibbonClient(name = "user", configuration = UserConfiguration.class)
```

Per completare la corretta configurazione di Ribbon, viene introdotto poi un bean “RestTemplate”, ovvero un’entità che permette di effettuare richieste http in ambiente Spring: tale template dovrà essere iniettato (@Autowired) da Spring ed essere marcato con

l'annotazione "@LoadBalanced", che va a specificare che questo bean sfrutterà le funzionalità di Load Balancing.

```
@Autowired
private RestTemplate template;


@Bean
@LoadBalanced
public RestTemplate template() {
    return new RestTemplate();
}
```

Infine, per poter effettuare chiamate con bilanciamento di carico, sarà sufficiente innescare una richiesta sul RestTemplate all'URL del servizio a cui vogliamo accedere (vedremo in seguito come è possibile configurarlo).

```
ResponseEntity<String> response = template
  .postForEntity(serverUrl, requestEntity, String.class);
return response.getBody();
```

Strategia di Load Balancing

Ribbon utilizza a default la strategia RoundRobin per effettuare il Load Balancing, ma è possibile configurare una qualsiasi altra strategia, sia tra quelle disponibilil sopracitate (Availability Filtering Rule e Weighted Response Time Rule), sia strategia custom. Questo è il compito del parametro che identifica la classe di configurazione del Load Balancer.

```
@Configuration
public class UserConfiguration {

    @Bean
    public IRule ribbonRule() {
        return new MyRule();
    }

}
```

I componenti per definire una configurazione del Load Balancer sono principalmente tre:

- Rule: componente logico che specifica la strategia di load balancing da attuare

- Ping: componente che specifica il meccanismo per determinare quando un server è pronto a rispondere ed operare in real-time.
- ServerList: oggetto statico o dinamico che contiene una lista di server correntemente attivi.

In una classe di configurazione è possibile specificare quali siano le regole da seguire per ciascuno dei tre componenti elencati sopra. Nel nostro caso, è sufficiente specificare il bean che si occupa della strategia di Load Balancing, definito da un'interfaccia *IRule*. Il metodo *ribbonRule* determina il tipo di load balancing che vogliamo avere, che può essere sia una classe di sistema, ad esempio RoundRobinRule, sia una classe custom che implementa l'interfaccia IRule oppure estende una classe di sistema già esistente.

```
public class MyRule implements IRule{

    private ILoadBalancer lb;

    @Override
    public Server choose(Object key) {
        List<Server> servers = lb.getAllServers();
        Random r = new Random();
        int rand = r.nextInt(100);
        System.out.println("Random number: " + rand);
        if (rand<50)
            return getServerByPort(servers,8090);
        else if(rand>=33 && rand <66)
            return getServerByPort(servers, 9999);
        return getServerByPort(servers, 8092);
    }

    private Server getServerByPort(List<Server> servers, int port){

    public void setLoadBalancer(ILoadBalancer lb) {

    public ILoadBalancer getLoadBalancer() {

}
```

In ogni caso, il metodo entry point per la strategia di load balancing è *choose*, il quale seleziona tra una lista di server quello che prenderà in carico la prossima richiesta. Nell'esempio mostrato, viene implementata una banale random choice, in cui la lista di server è statica e definita a priori.

Eureka

Nel caso in cui i server non siano prefissati a priori ma siano dinamicamente scoperti, come succede spesso nei reali sistemi distribuiti, sia in caso di guasti, sia in caso di upgrade di nodi servitori, sia in caso di riassegnamento di URL, la soluzione sopra proposta non funziona più.

Eureka risponde in maniera semplice e completa a questa esigenza: offre, infatti, un servizio di discovery, in cui le applicazioni vive nel sistema distribuito possono registrarsi, sia client che server, per agevolare la ricerca runtime delle entità presenti.

Configurazione e funzionamento

Innanzitutto, è necessario configurare nel file *pom.xml* le giuste dipendenze.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

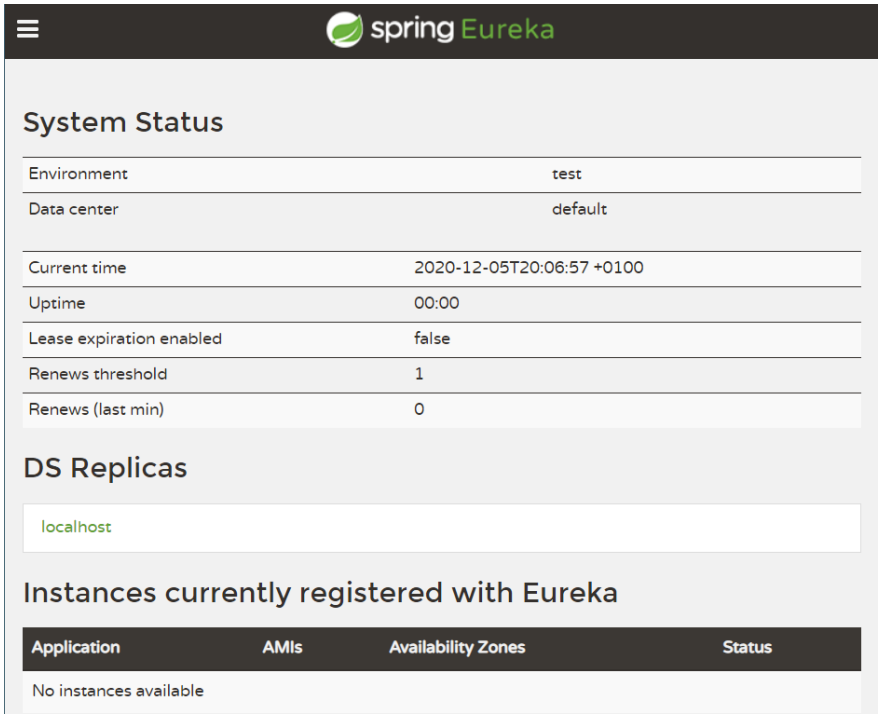
A default, la porta in cui il discovery server verrà messo in ascolto è la 8761, ma può essere configurata negli appositi file di configurazione (application.properties o application.yml). La semplicità di Eureka fa sì che, con una sola annotazione, "@EnableEurekaServer", si possano già attivare tutte le funzionalità di tale applicazione.

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryApplication {

    public static void main(String[] args) {
        SpringApplication.run(DiscoveryApplication.class, args);
    }

}
```

Una volta messo in esecuzione, riceverà ogni richiesta di registrazione di clienti o servizi e li renderà accessibili a chiunque sia registrato a sua volta sul discovery server. Se accediamo all'URL http://localhost:8761 è possibile vedere una lista delle istanze correntemente registrate su Eureka.



Zuul

Finora abbiamo capito come riuscire a creare un sistema distribuito con client-side load balancing, con una gestione runtime del discovery dei servizi attivi. Vediamo adesso come sviluppare un'applicazione distribuita con server-side load balancing.

Zuul è un'applicazione front-door per tutte le richieste che vengono effettuate dai clienti, verso il backend dei servizi messi a disposizione. Il routing è una delle parti più importanti in un'architettura a microservizi, infatti permette di poter raggiungere dall'esterno tutti i microservizi presenti nel sistema. Zuul utilizza una vasta gamma di tipi di filtri, che ci consentono di applicare rapidamente e agilmente funzionalità al nostro edge-service. Tali filtri aiutano a svolgere funzioni di autenticazione e sicurezza, monitoraggio, routing dinamico.

Zuul è costituito da quattro componenti principali:

- **Zuul-core:** libreria che contiene le funzionalità core di compile e esecuzione dei filti.
- **Zuul-simple-webapp:** webapp che mostra un semplice esempio di utilizzo e di come fare la build di un'applicazione con zuul-core.
- **Zuul-netflix:** libreria che espone interfacce di comunicazione con tutti gli altri componenti di Netflix OSS (eureka, ribbon…).

- **Zuul-netflix-webapp:** webapp che unisce zuul-core e zuul-netflix in un unico componente facile da usare.

Configurazione e funzionamento

Come tutti gli altri componenti che sono stati illustrati, la prima cosa da fare per poter sfruttare zuul all'interno di un sistema distribuito è configurare le dipendenze nel file *pom.xml*.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

Oltre alla configurazione delle dipendenze, è possibile sfruttare le funzionalità di routing di zuul tramite il file *application.properties* oppure *application.yml*.

```
zuul.routes.user.path=/user/**
zuul.routes.user.path.serviceId=client-service
zuul.routes.user.path.stripPrefix=true
```

In questo caso, stiamo definendo una route per un servizio *client-service* tramite il quale, per poter accedere a tale servizio, è sufficiente porre il suffisso *user* all'URL in cui zuul è in esecuzione. Per esempio, se zuul è in esecuzione in localhost:8662, per poter accedere al servizio *client-service* è sufficiente fare una richiesta (esistente nel servizio stesso) al path *localhost:8662/user/operazione*.

L'altra funzionalità di grande interessa è il server-side load balancing, configurabile nello stesso file in questione: è infatti possibile specificare una classe Load Balancer che incapsula la strategia di load balancing desiderata.

```
detection.ribbon.NFLoadBalancerRuleClassName=zuul.proxy.RandomWeightedRoundRobin
```

Per mettere in esecuzione zuul, è necessario definire una classe con diverse annotazioni:

- @RibbonClient: stesso principio di funzionamento di come spiegato nella sezione su Ribbon, in cui specifichiamo una classe di configurazione. Questo attiva l'utilizzo di server-side Load Balancer.
- @EnableEurekaClient: abilita la registrazione e il pull di istanze esistenti e registrate su eureka.
- @EnableZuulProxy: annotazione specifica di Zuul, che rende possibile abilitare tutti i funzionamenti citati in questa sezione.

```
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
@RestController
@RibbonClient(name = "detection", configuration = UserConfiguration.class)
public class ZuulProxy {

    public static void main(String[] args) {
        SpringApplication.run(ZuulProxy.class, args);
    }

    @RequestMapping
    public String get() {
        return "ZUUL PROXY";
    }

}
```

Osservazioni tecnologia

Il massiccio utilizzo di questa tecnologia, Spring Cloud, comporta diversi vantaggi e svantaggi rispetto ad altri framework esistenti. Primo fra tutti, tra gli svantaggi, c'è il fatto che il consumo di banda può risultare più cospicuo in tecnologia Spring Cloud, poiché si utilizza il protocollo HTTP con, generalmente, messaggi JSON, che consumano di più rispetto a soluzioni in cui il trasferimento è binario (Apache Dubbo). Al tempo stesso, tra i vantaggi più notevoli si ha la facilità di development e configurazione di Spring Cloud, i quali, sempre comparati a Dubbo, vengono molto semplificati. In Spring Cloud i vincoli di interfaccia sono molto liberi, il che può potenzialmente portare ad aggiornamenti delle interfacce in modo non consono al contesto di utilizzo; questo si risolve con una forte competenza amministrativa affiancata allo sviluppo. Spring Cloud può utilizzare, come registration service, solo Eureka oppure una "self-research",

mentre altre tecnologie, come Dummo, supportano utilizzo di Zookeeper, piuttosto che Redis. Un fattore da non sottovalutare è il lavoro e il continuo mantenimento dei framework in questione, in cui Spring Cloud è leader. Per esempio, Dummo non è mantenuto né costantemente aggiornato, al contrario di Spring Cloud. Un altro punto cruciale è l'utilizzo maggiore di Spring Cloud: sviluppatori che accolgono tale tecnologia saranno molto più probabilmente inseriti in ambienti lavorativi reali.

Un vero e proprio collo di bottiglia potrebbe essere il non corretto deployment delle varie istanze dei servizi di cui necessitiamo, per esempio Zuul. Essendo un single point of entry, se dovessimo scalare su un insieme molto grande di nodi, diventerebbe un bottleneck. Come potremmo ovviare a questa cosa? Una soluzione potrebbe essere quella di creare molte istanze di Zuul in nodi distinti, ma perderemmo la potenzialità dei route della tecnologia. Un esempio di applicazione di questo scenario è di registrare istanze multiple e metterle dietro a un cloud-based load-balancer, come per esempio Amazon Web Service – Application Load Balancer (AWS ALB).

Come ultima considerazione, è necessario sottolineare che in un'elevata percentuale di casi, vicino al 100%, il consumo di rete di Spring Cloud, rispetto a quello di Dummo, non è un grosso problema, e, se dovesse diventarlo, è molto facile adottare strategie di compressione, di segmentazione o di caching che permettano di attenuare i consumi senza rinunciare ad utilizzare tale framework.

Progetto

Il progetto che è stato sviluppato consiste nell'implementazione di un servizio clusterizzato in tre nodi distinti, che offre una funzionalità di Object Recognition data un'immagine. A tal fine è stata implementata un'applicazione Spring Boot server-side che offrisse la logica di business, della quale è stato fatto il deploy nei tre nodi suddetti. È stata poi sviluppata un'applicazione Zuul, che permettesse la possibilità di utilizzare diverse strategie di server-side load balancing, oltre all'essenziale routing delle richieste. Lato client, si dispone di una semplice applicazione Spring Boot che effettua una richiesta al nodo Zuul per il servizio di Object Recognition. Tutte le istanze in esecuzione sono registrate in un discovery Eureka, al fine di permettere una gestione ottimizzata delle entità vive runtime e real-time.

Discovery

Il servizio di discovery, realizzato grazie ad Eureka, è stato configurato per essere messo in esecuzione all'url 137.204.57.68, con porta 8761. La classe che incapsula il main dell'applicazione non differisce dall'esempio mostrato nella sezione su Eureka.

```
server.port=8761

spring.application.name=discovery-server

eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
eureka.client.serviceUrl.defaultZone=http://137.204.57.68:8761
```

Client

L'applicazione Client consiste in una pagina web che offre la possibilità di caricare un'immagine a scelta (dimensione massima 20MB) e di richiederne l'Object Recognition verso il servitore. La configurazione è stata eseguita impostando come nome "user", con registrazione al discovery Eureka e con il supporto a Feign, componente per l'effettuazione di chiamate REST verso il server.

```
eureka.client.serviceUrl.defaultZone: http://137.204.57.68:8761/eureka/
server.port=8082
spring.application.name=user
feign.client.config.default.connectTimeout: 160000000
feign.client.config.default.readTimeout: 160000000
```

La main app non deve far altro che abilitare l'Eureka client e il Feign client.

```java
@SpringBootApplication
@EnableFeignClients
@EnableEurekaClient
public class UserApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class, args);
    }

}
```

Server

Il server è un'applicazione che sfrutta le librerie di tensorflow per effettuare il recognition di immagini. La configurazione è, anche in questo caso, semplice: è sufficiente, infatti, dare un nome al servizio (in questo caso *detection*) e configurare il discovery Eureka.

```yaml
spring:
  application:
    name: detection
  servlet:
    multipart:
      max-file-size: 20MB
      max-request-size: 30MB

eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
    leaseExpirationDurationInSeconds: 30
  client:
    serviceUrl:
      defaultZone: http://137.204.57.68:8761/eureka/
server:
  port: 8081
```

Per effettuare correttamente l'object recognition, è stato predisposto un controller, chiamato UploadController, il quale si occupa di ricevere il file, salvarlo temporaneamente in una cartella locale, effettuare la prediction sull'immagine e restituire il best match di object recognition.

```java
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RestController
@RequestMapping("/appliance")
public class UploadController {

    private static String UPLOADED_FOLDER = "/home/debian/";

    public String predict() {

}
```

Routing e Load Balancing

Infine, è stato implementato il meccanismo di routing e load balancing tramite Zuul, anch'esso registrato sul discovery Eureka. È stato configurato per lavorare nella porta 8662, con nome del servizio *gateway-service* e con due route, una per i server e una per il client.

```
eureka.client.serviceUrl.defaultZone: http://137.204.57.68:8761/eureka/
eureka.client.fetchRegistry=true
eureka.instance.hostname=137.204.57.68

server.port=8662
spring.application.name=gateway-service

zuul.routes.detection.path=/predict/**
zuul.routes.detection.serviceId=detection
zuul.routes.detection.stripPrefix=true
detection.ribbon.eureka.enabled=true

zuul.routes.user.path=/user/**
zuul.routes.user.path.serviceId=user
zuul.routes.user.path.stripPrefix=true

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds= 1000000
hystrix.command.default.execution.isolation.strategy= THREAD
ribbon.ReadTimeout=60000

detection.ribbon.NFLoadBalancerRuleClassName=zuul.proxy.RandomWeightedRoundRobin
```

```java
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
@RestController
@EnableDiscoveryClient
@RibbonClient(name = "detection", configuration = UserConfiguration.class)
public class ZuulProxy {

    public static void main(String[] args) {
        SpringApplication.run(ZuulProxy.class, args);
    }

    @RequestMapping
    public String get() {
        return "ZUUL PROXY";
    }

}
```

Per poter accedere all'operazione di predict passando da zuul, si dovrà quindi effettuare una richiesta all'URL:

```
137.204.57.68:8662/predict/appliance/predict
```

Osservazioni soluzione

La soluzione adottata sfrutta un nodo accessibile con IP pubblico che contiene sia l'Eureka discovery, sia Zuul sia un'istanza di servizio di object recognition. Sfruttare questo tipo di soluzione è stato necessario nella configurazione predisposta, in quanto un cliente esterno deve poter sapere dove è Zuul, quindi deve accedere ad Eureka, e poi comunicare con Zuul. Inoltre, avendo solo tre nodi a disposizione, è stato necessario inserire un'istanza del servizio finale anche nel nodo in questione, in quanto fare testing solo su due nodi non avrebbe portato a risultati più vicini ad un ambiente reale. Come citato nel capitolo precedente, un possibile collo di bottiglia di Spring Cloud potrebbe essere il non corretto deployment delle varie istanze nei nodi. Effettivamente, avendo messo tre istanze di tre servizi diversi sullo stesso nodo, può portare, se scaliamo il numero di richiesta, ad un rallentamento o addirittura crash di almeno una di queste istanze. Infatti, nei vari test effettuati, talvolta si hanno crash di un servizio a fronte di numerose richieste. Ovviamente, se ci fossero stati altri nodi disponibili, si sarebbe

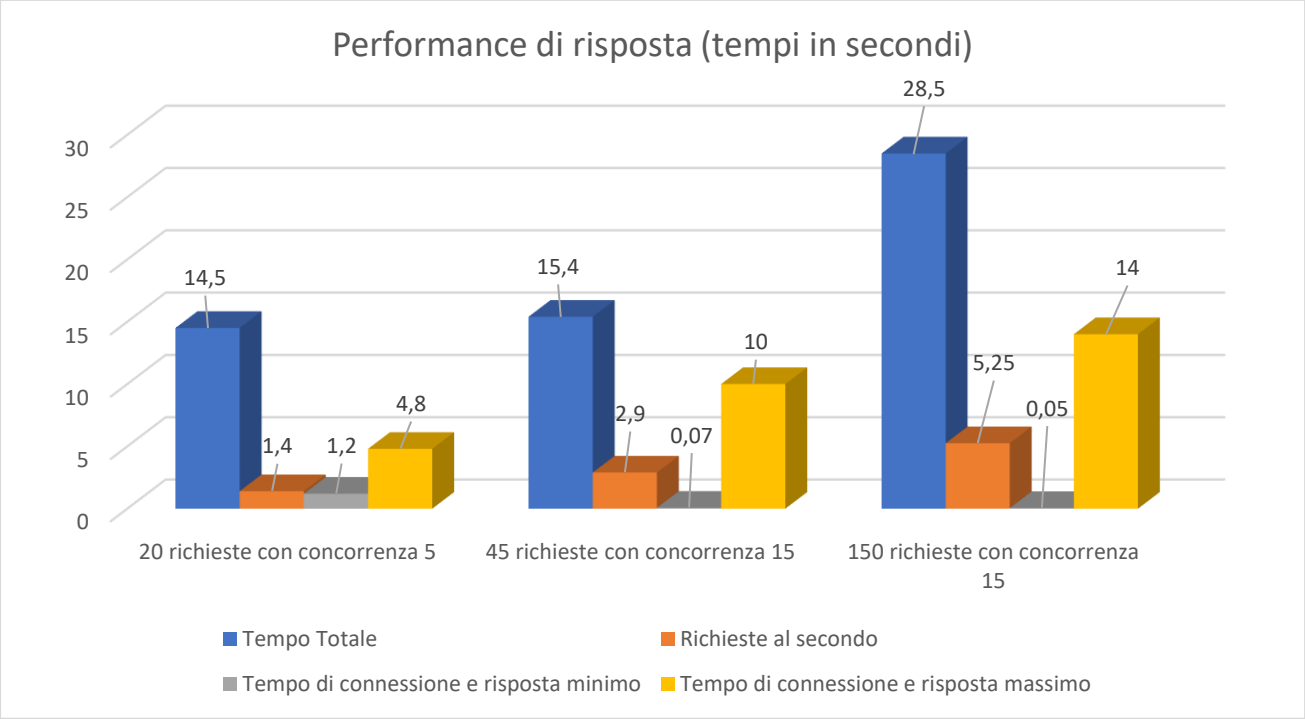
potuto fare un deployment in cui venivano sfruttati cinque nodi, ognuno con un'istanza di un servizio. Le performance, che saranno analizzate nella prossima sezione, sono comunque risultate buone ed è stato possibile misurarle come se fossimo in un ambiente pseudo-reale.

Performance

La misurazione delle performance del sistema distribuito creato è stata effettuata grazie all'utilizzo di tre macchine virtuali, gentilmente concesse da unibo. Una di queste tre macchine è raggiungibile dall'esterno con IP pubblico 137.204.57.68, e per questo è stata usata sia per mettere in esecuzione il discovery, sia lo zuul proxy, oltre ad un'istanza del server. Altre due macchine, con indirizzi IP locali 192.168.60.34 e 192.168.60.91, sono state usate per rendere vive altre due istanze di server, per poter analizzare il comportamento e l'efficienza del load balancing effettuata da Zuul. Le macchine virtuali utilizzate hanno come configurazione: 2 CPU Intel Xeon E3-12xx v2 a 2.3 GHz, 4 GB di memoria RAM e 40 GB di spazio di archiviazione. Di seguito sono riportate tutte le misurazioni, suddivise per strategia di load balancing implementata.

Round Robin

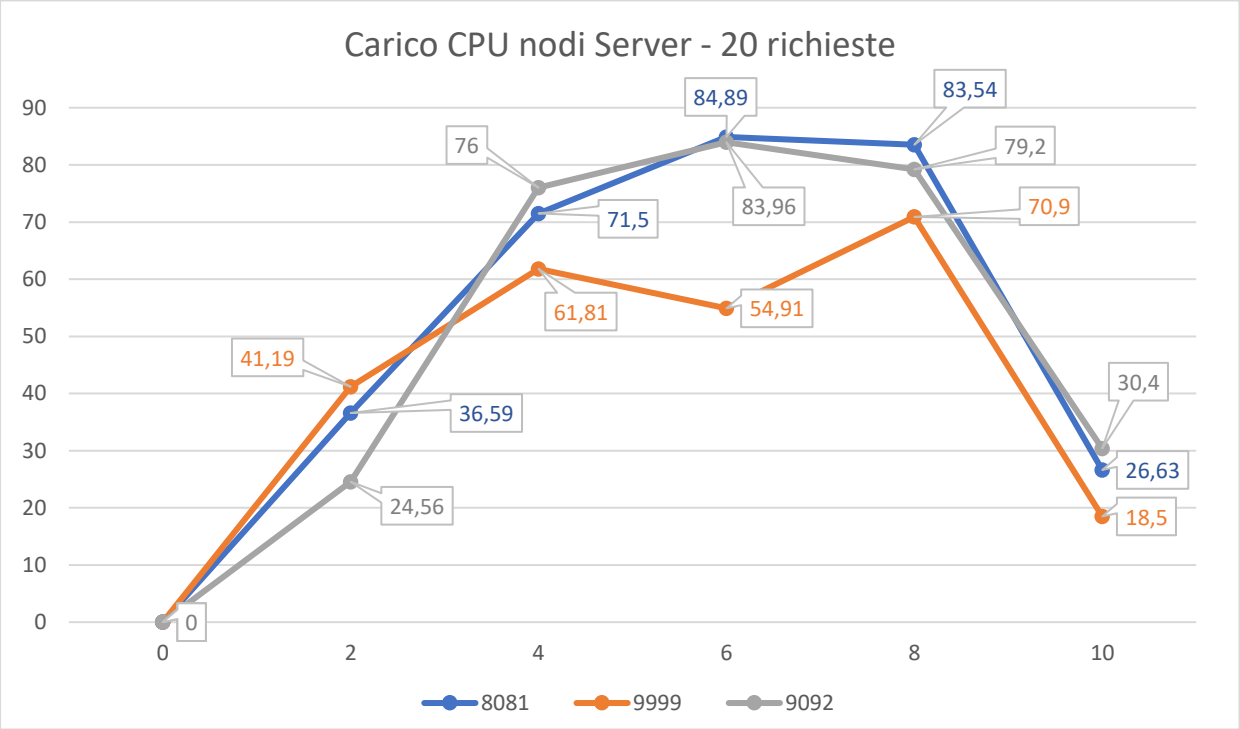
La strategia di load balancing Round Robin è la più semplice e intuitiva, nonché il default. Viene calcolato un contatore in modulo tre e si passa la richiesta corrente al nodo successivo, sfruttando un'assegnazione ciclica, con una richiesta a nodo alla volta.



Percentuale richieste fallite:

- 20 richieste → 0%
- 45 richieste → 22%
- 150 richieste → 66%

Carico e uso CPU 20 richieste:



Average uso CPU:

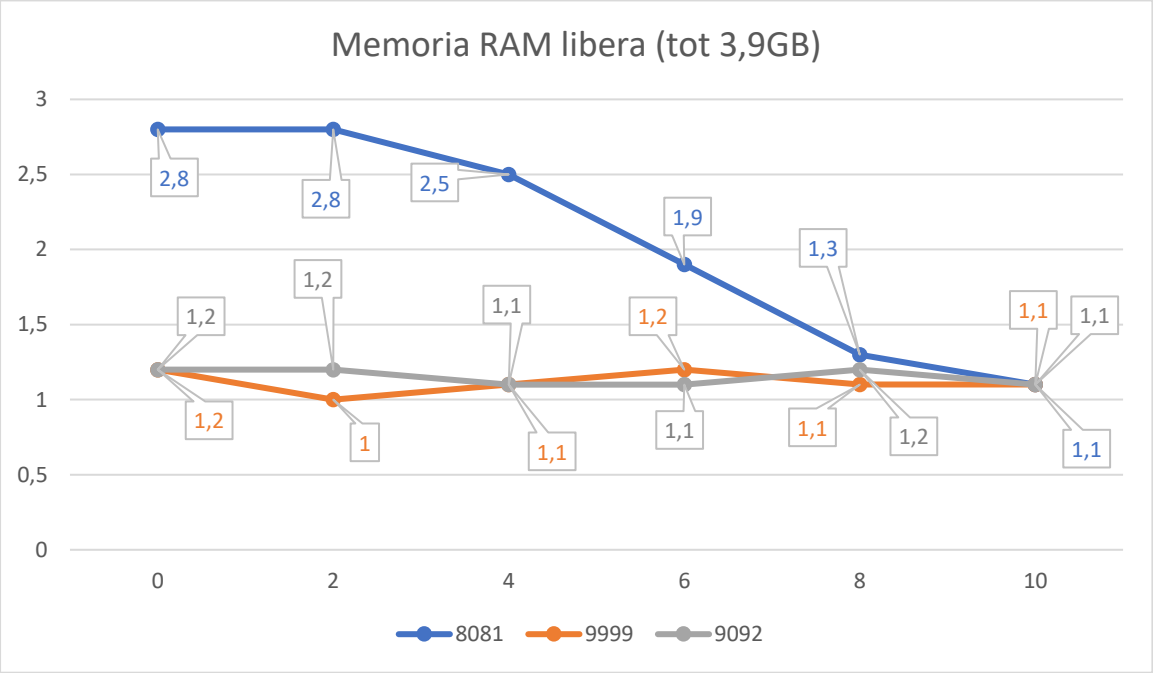
- 8081 → 60,63 %
- 9999 → 49,5 %
- 9092 → 58,8 %

Discostamento dalla media:

- 8081 → 60,63-56,31 = 4,32
- 9999 → 56,31 – 49,5 = 6,81
- 9092 → 58,8 – 56,31 = 2,49

Discostamento medio: 4,54

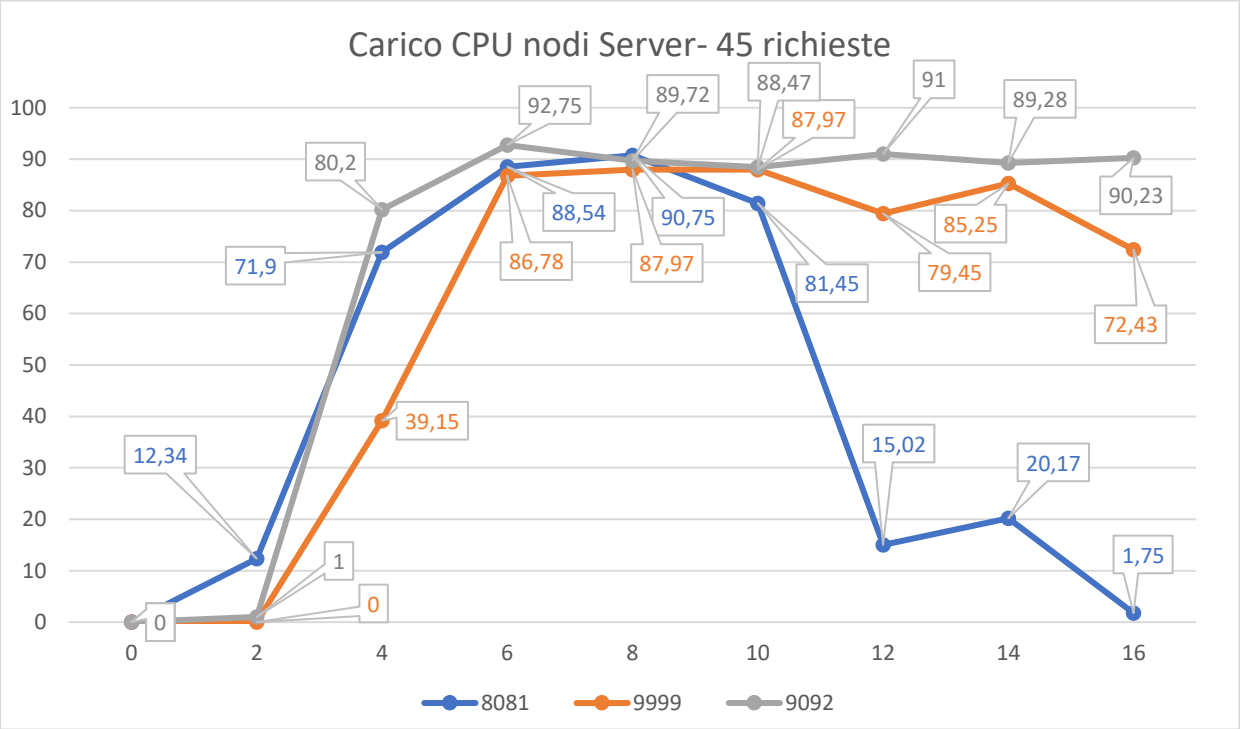
Consumo memoria RAM 20 richieste:



Average memoria utilizzata:

- 8081 → 47 %
- 9999 → 71,5 %
- 9092 → 70,5 %

Carico e uso CPU 45 richieste:



Average uso CPU:

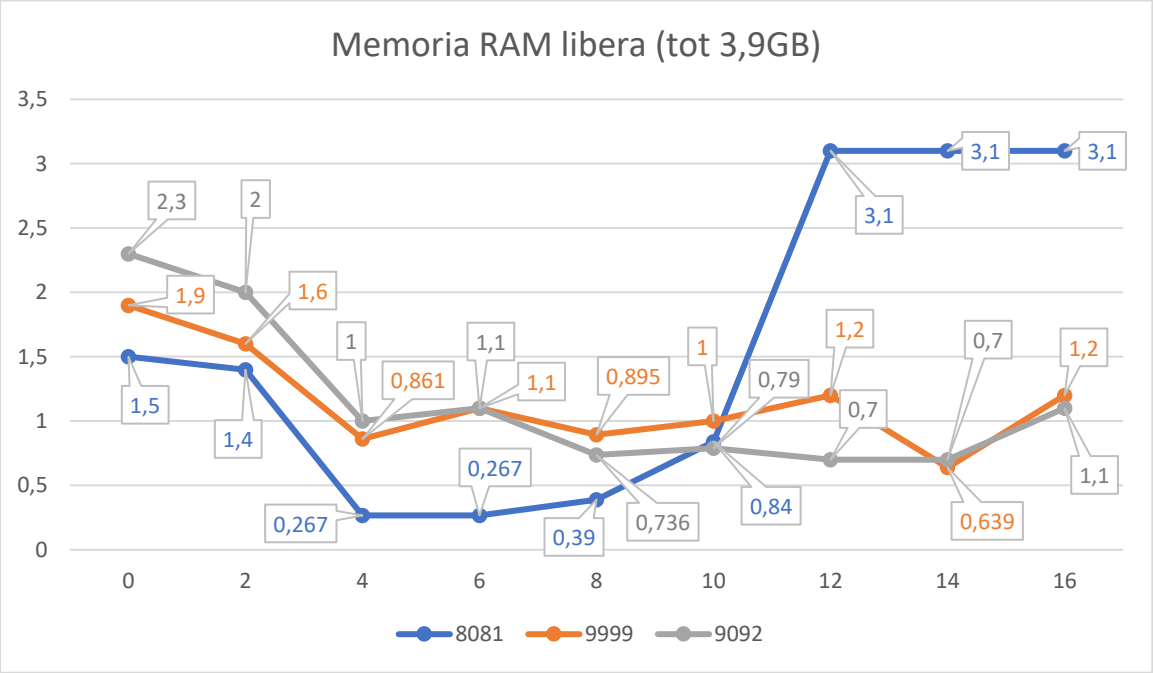
- 8081 → 47,74 %
- 9999 → 67 %
- 9092 → 77 %

Discostamento dalla media:

- 8081 → 63,91 – 47,74 = 16,17
- 9999 → 67 – 63,91 = 3,09
- 9092 → 77 – 63,91 = 13,09

Discostamento medio: 10,78

Consumo memoria RAM 45 richieste:



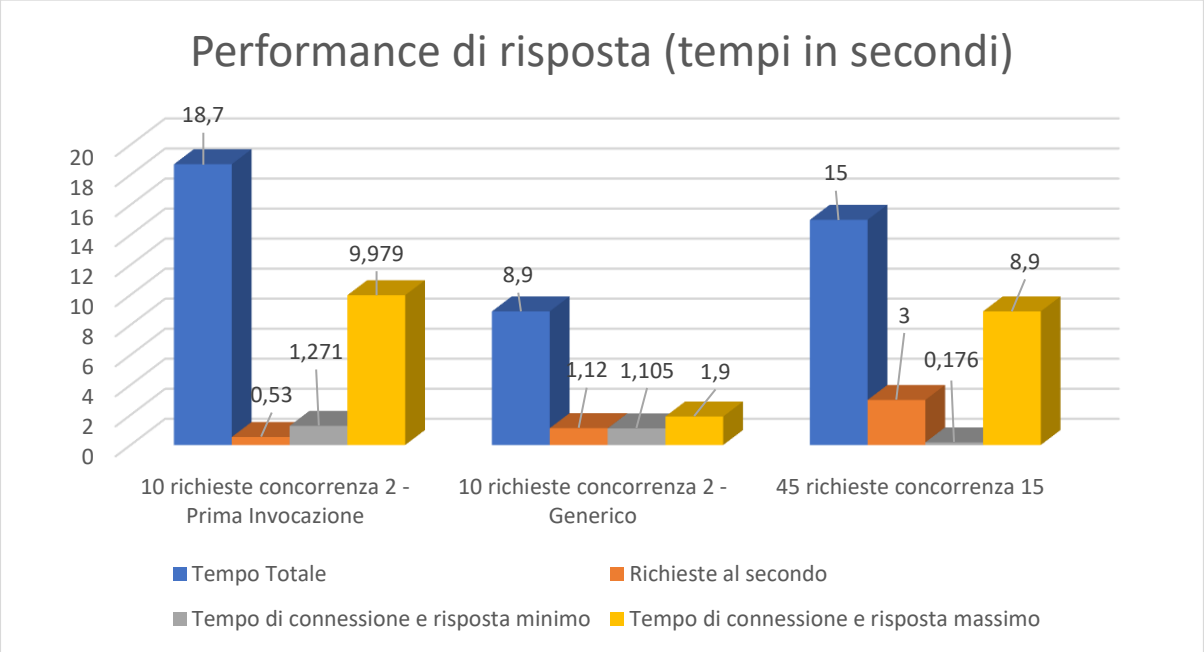
Average memoria utilizzata:

- 8081 → 60 %
- 9999 → 70,5 %
- 9092 → 70,5 %

A causa dell'elevata percentuale di richieste fallite causa crash dei nodi server, non vengono riportate le statistiche di utilizzo risorse relative a 150 richieste.

Weighted Response Time

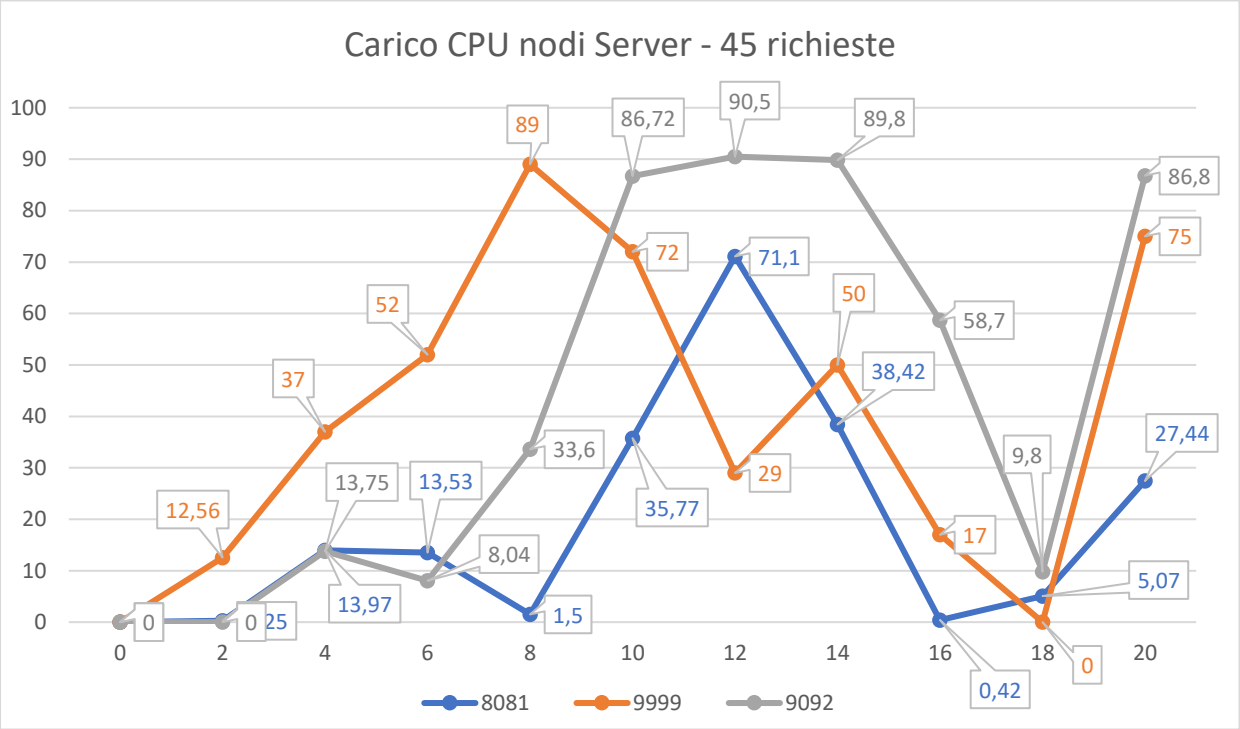
Strategia nella quale viene analizzato il tempo di risposta medio di un nodo, viene messo a confronto con gli altri nodi e a ciascuno di essi viene associato un peso, proporzionato al tempo di risposta calcolato. In questi casi, la prima invocazione è ancora più lunga, dovuta al calcolo del tempo di risposta e dei pesi.



Percentuale richieste fallite:

- 10 richieste prima invocazione → 0%
- 10 richieste generico → 0%
- 45 richieste → 37%

Carico e uso CPU 45 richieste:



Average uso CPU:

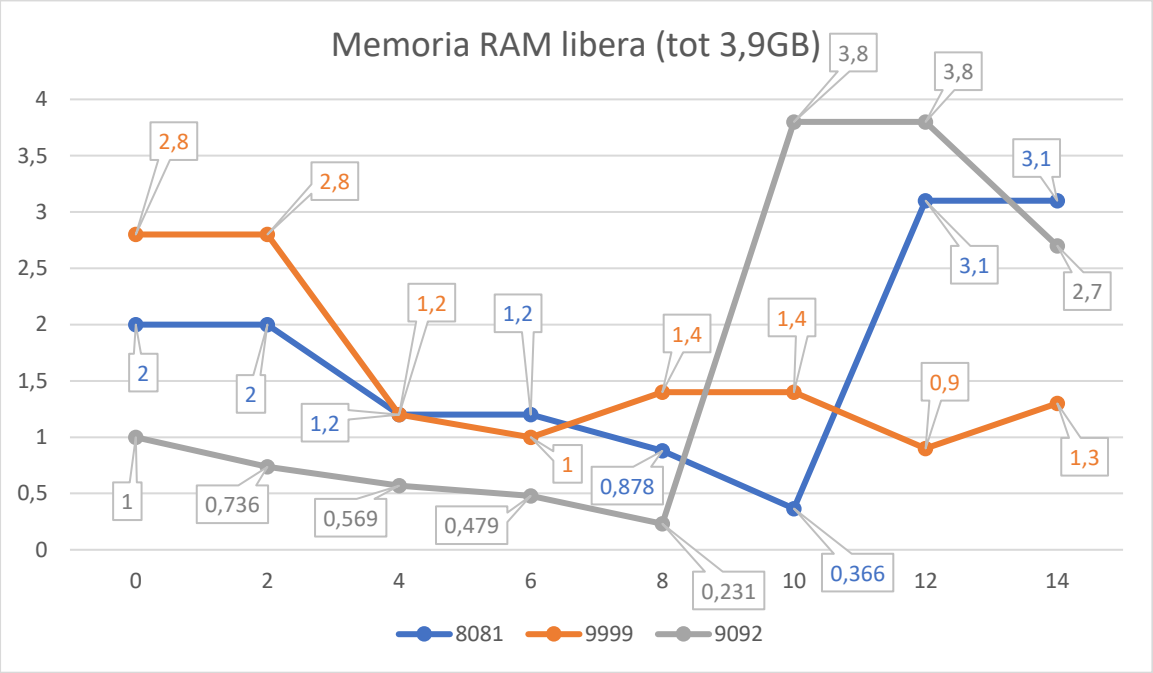
- 8081 → 20,7 %
- 9999 → 43,3 %
- 9092 → 47,7 %

Discostamento dalla media:

- 8081 → 37,23 – 20,7 =16,53
- 9999 → 43,3 – 37,23 = 6,07
- 9092 → 47,7 – 37,23 = 10,47

Discostamento medio: 11,02

Consumo memoria RAM 45 richieste:

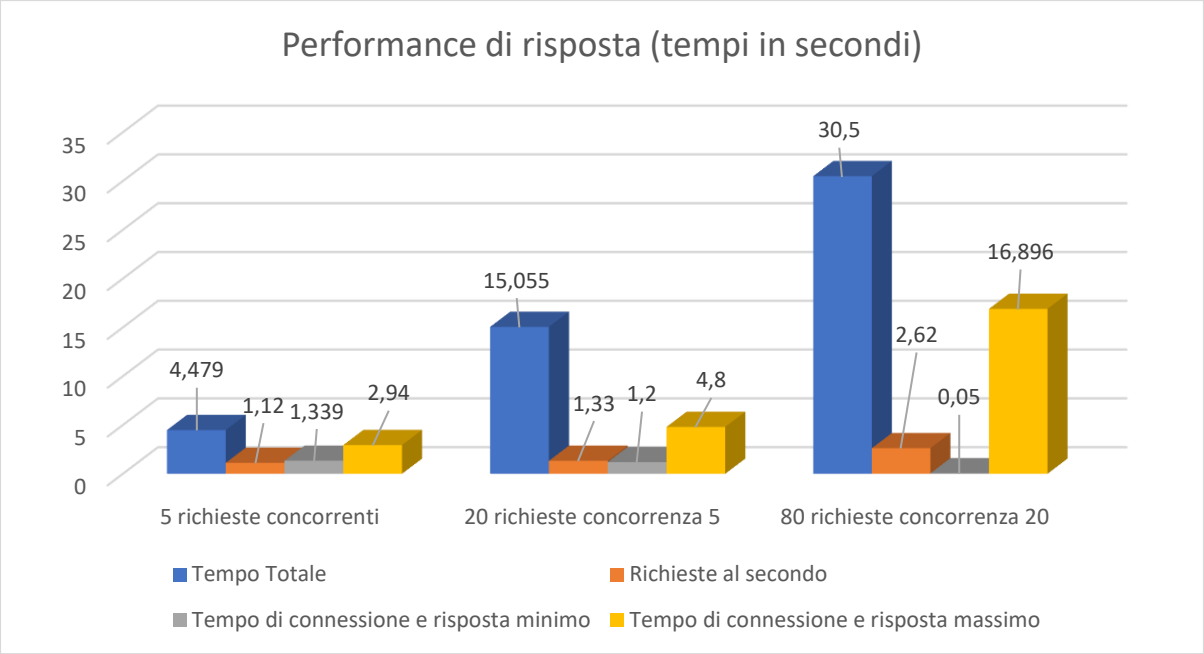


Average memoria utilizzata:

- 8081 → 55,6 %
- 9999 → 59 %
- 9092 → 57,4 %

Random Server

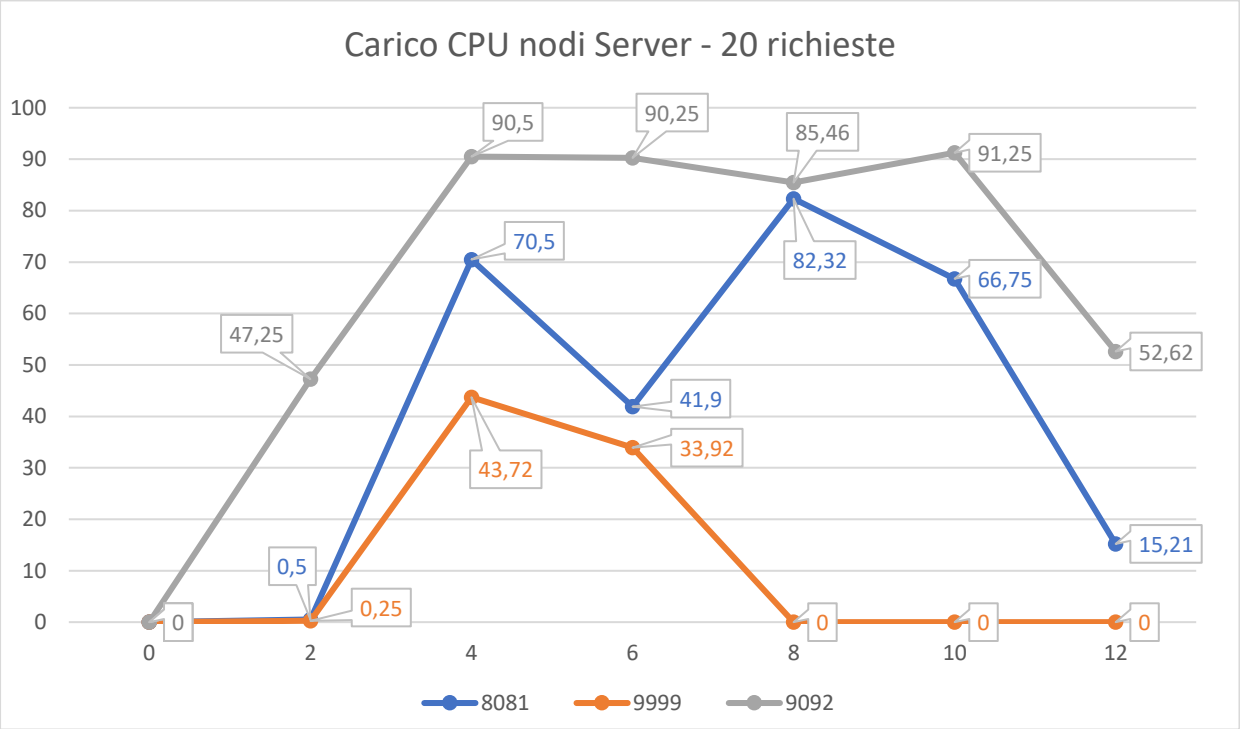
Strategia tramite la quale si genera un numero casuale in modulo al numero dei nodi server e si redirige la richiesta al nodo estratto.



Percentuale richieste fallite:

- 5 richieste → 0%
- 20 richieste → 0%
- 80 richieste → 42%

Carico e uso CPU 20 richieste:



Average uso CPU:

- 8081 → 46,2 %
- 9999 → 12,3 %
- 9092 → 76,2 %

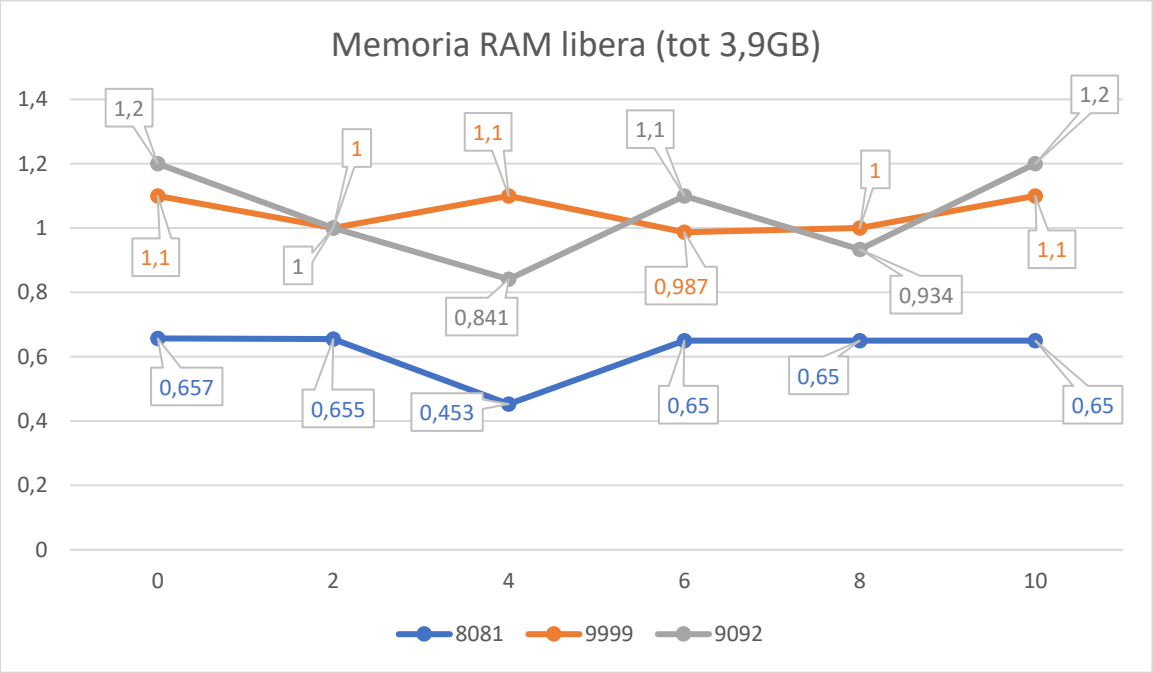
Discostamento dalla media:

- 8081 → 46,2 – 44,9 = 1,3
- 9999 → 44,9 – 12,3 = 32,6
- 9092 → 76,2 – 44,9 = 31,3

Discostamento medio: 21,7

Osservazione: strategia peggiore, carichi dei server molto diversi.

Consumo memoria RAM 45 richieste:

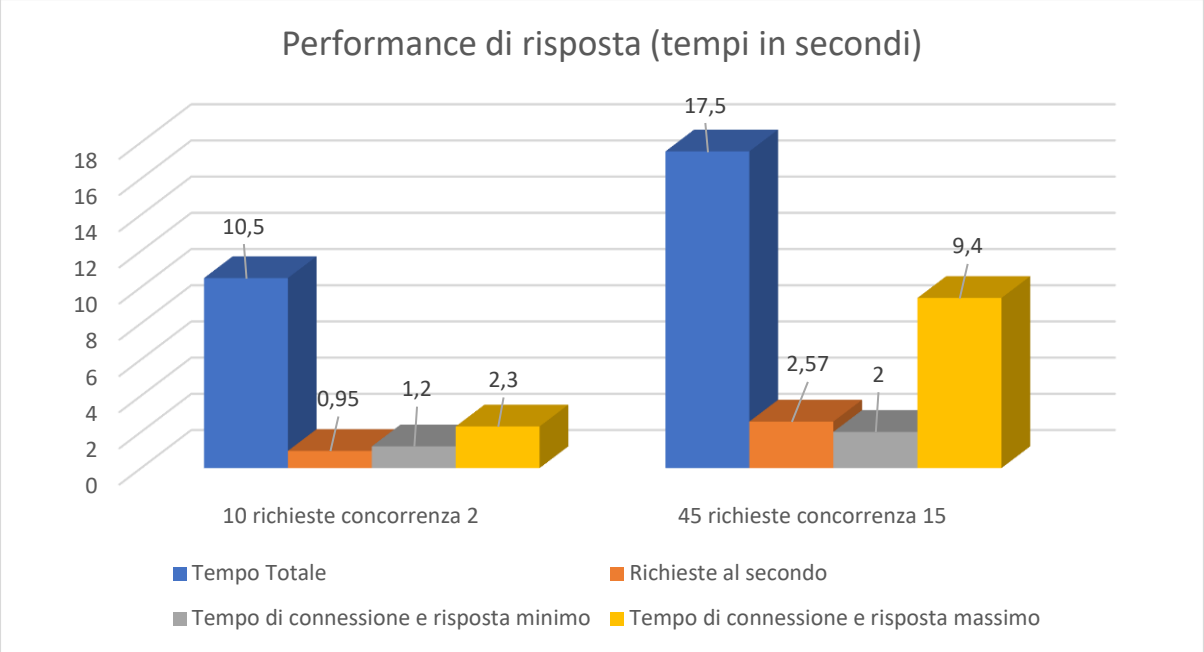


Average memoria utilizzata:

- 8081 → 84 %
- 9999 → 71,2 %
- 9092 → 73,1 %

Random Weighted Round Robin

Strategia custom che implementa una variante di Weighted Round Robin, in cui a ciascun nodo viene associato un peso, che sarà il numero di richieste da servire prima di passare al nodo successivo. In questa strategia si sfrutta un meccanismo in cui si associa in maniera casuale un peso a ciascun nodo e poi si segue la strategia Weighted Round Robin.



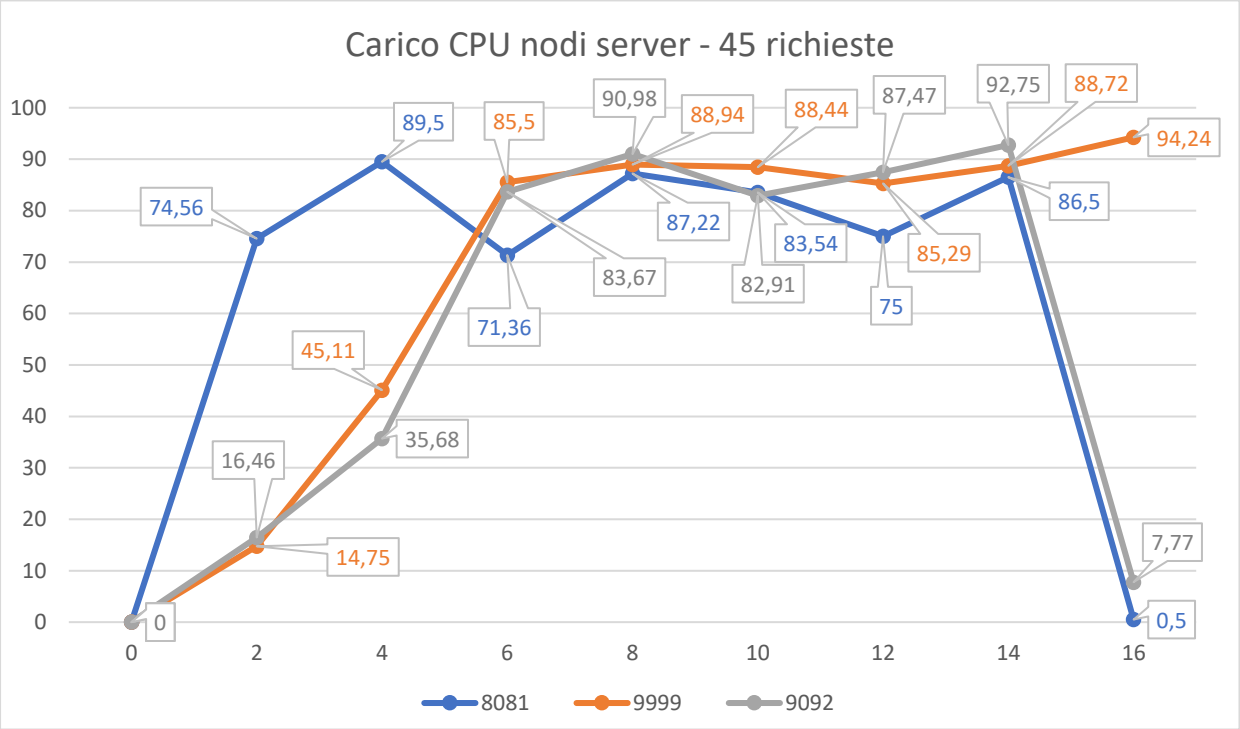
Percentuale richieste fallite:

- 10 richieste → 0%
- 45 richieste → 0%
- 150 richieste → 93%

Osservazione: il dato relativo a 150 richieste con concorrenza 15 non è stato riportato, in quanto la percentuale di richieste fallite è elevatissima.

```
Concurrency Level:      15
Time taken for tests:   14.455 seconds
Complete requests:      150
Failed requests:        140
   (Connect: 0, Receive: 0, Length: 140, Exceptions: 0)
Non-2xx responses:      140
```

Carico e uso CPU 45 richieste:



Average uso CPU:

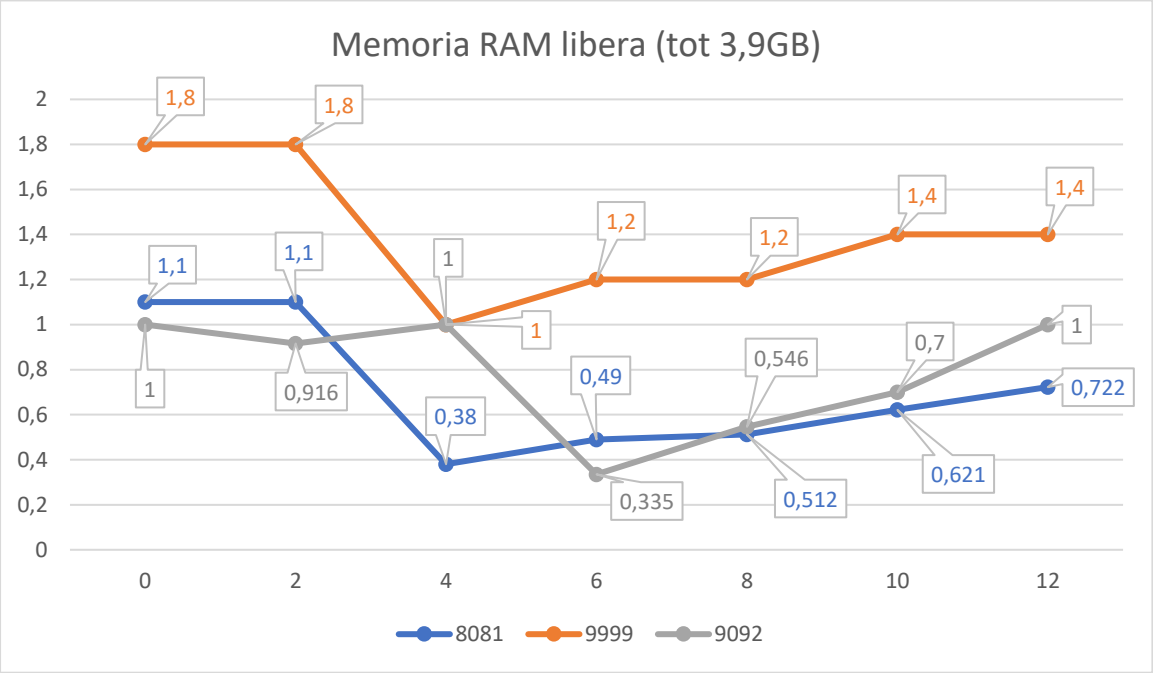
- 8081 → 71 %
- 9999 → 73,8 %
- 9092 → 62,2 %

Discostamento dalla media:

- 8081 → 71 – 69 = 2
- 9999 → 73,8 – 69 = 4,8
- 9092 → 69 – 62,2 = 6,8

Discostamento medio: 4,5

Consumo memoria RAM 45 richieste:



Average memoria utilizzata:

- 8081 → 82 %
- 9999 → 64 %
- 9092 → 80 %

Considerazioni

Analizzando i dati illustrati nelle sezioni precedenti, possiamo fare un confronto tra le varie strategie di load balancing implementate. Innanzitutto, la strategia che è risultata peggiore è quella del Random Server, come era prevedibile. Infatti, lo scostamento dalla media del carico dei nodi server è di ben 21.7, dato non ottimo, che fa notare la non indifferente varietà dei carichi dei nodi server. Weighted Response Time e Round Robin hanno raggiunto circa lo stesso livello di performance, se non fosse che la prima strategia è molto più lenta nella prima invocazione, a causa del calcolo del tempo di risposta e dei pesi dei vari nodi. Il discostamento della media di carico dei nodi è rispettivamente di 11.02 e 10.78, nel caso di 45 richieste con concorrenza 15. La strategia risultata migliore nell'insieme di statistiche misurate in questo progetto è stata la Random Weighted Round Robin, con uno scostamento relativo a 45 richieste di solo 4.5, un ottimo risultato.

Un'osservazione che possiamo fare è relativa al consumo di memoria RAM: non sono stati rilevati consumi anomali di RAM, ma è stato visto sperimentalmente che, quando la memoria libera scendeva sotto i 300/200 MB, l'istanza di quel nodo andava in crash. Infatti, nel caso di Weighted Response Time con 45 richieste simulate, si può notare come il nodo 8081 scende sotto i 300 MB di memoria libera e subito dopo va in crash, riportando la memoria occupata a livelli bassi, insieme al consumo di CPU. Il consumo più uniforme di memoria RAM è stato rilevato, stranamente, nel Random Server, ma anche nel Random Weighted Round Robin.

Conclusioni

Spring Cloud è una tecnologia in continua evoluzione, ma che ha già incarnato moltissimi aspetti dei sistemi distribuiti, offrendo soluzioni ad hoc in svariati casi di utilizzo. La sua estrema flessibilità lo portano ad essere usato molto spesso, sia per sfruttare soluzioni standardizzate nel framework, sia per creare, con relativa facilità, nuove strategie custom per vari scenari. Purtroppo, Spring Cloud Load Balancer non offre ancora funzionalità server-side né un modo efficiente per implementare nuovi algoritmi, ma l'architettura di Spring Cloud che si appoggia a Netflix OSS è ben fatta (Spring Cloud Netflix). Si riesce a creare un sistema distribuito con le principali funzionalità proprie di esso, load balancing, affidabilità, scalabilità, in maniera efficace ed efficiente. La realizzazione di questo progetto ha rafforzato la conoscenza che sta alla base sia delle strategie di load balancing, sia alle altre caratteristiche proprie dei sistemi distribuiti. Infatti, toccare con mano un'applicazione distribuita, doverne fare il deployment in nodi distinti, doverne misurare le performance, notare, per esempio, che sotto un certo livello di RAM un nodo può andare in crash e tutte le altre esperienze dovute alla progettazione e sviluppo del sistema, aumentano la consapevolezza dei contesti nel mondo reale, aiutando a prevedere e prevenire determinati comportamenti a livello pratico, oltre che ad aumentare la fiducia nell'utilizzo delle tecnologie in questione.

Sitografia

- *Documentazione ufficiale Spring Cloud, https://spring.io/projects/spring-cloud*
- *Client-Side Load Balancer, https://spring.io/guides/gs/spring-cloud-loadbalancer*
- *Algoritmi di Load Balancing, https://www.ionos.it/digitalguide/server/know-how/load-balancer-bilanciamento-del-carico-nel-server*
- *Corso di Infrastructures for Cloud Computing and Big Data, https://www.unibo.it/it/didattica/insegnamenti/insegnamento/2019/421590*
- *Microservizi in Java con Spring Boot e Spring Cloud, https://codingjam.it/microservizi-in-java-con-spring-boot-e-spring-cloud*
- *Spring Cloud Config, https://cloud.spring.io/spring-cloud-config/reference/html*
- *Spring Cloud Ribbon, https://www.baeldung.com/spring-cloud-rest-client-with-netflix-ribbon*
- *Spring Cloud Netflix, https://spring.io/projects/spring-cloud-netflix*