ALMA MATER STUDIORUM-UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

Attività progettuale di Sistemi Distribuiti M

SISTEMA DISTRIBUITO MULTIMEDIALE CON RIBBON E EUREKA

Progetto di :
Dario Grandi

Relatore:
Prof.Ing.Paolo Bellavista

Anno accademico 2017/2018

1.0 Introduzione

Lo scopo di questa relazione è fornire una descrizione sullo sviluppo di un sistema distribuito utilizzando le tecnologie per il Cloud messe a disposizione da Netflix OSS e Spring Framework.
Le tecnologie di Netflix utilizzate per questo progetto sono Ribbon, per quanto riguarda il bilanciamento del carico e Eureka per il servizio di Service Discovery.
L'obbiettivo è quello di descrivere entrambi le tecnologie nel dettaglio e fornire un'implementazione sulla loro integrazione.
Inizialmente nella prima parte della relazione verrà descritto il concetto di service Disovery e Bilanciamento del carico, elencando i possibili approcci che si possono utilizzare al giorno d'oggi.
Verrà fornitò anche una descrizione su Netflix e le principali tenclogie create per il Cloud che mette a disposizione.
Successivamente si entrerà nel dettaglio delle tecnologie Ribbon e Eureka fornendo una descrizione dettagliata sul loro comportamento e la loro integrazione attraverso Spring.
Nella seconda parte della relazione verrà fornita un'implementazione del sistema che andrà ad utilizzare le tecnologie sopracitate e cercherà di analizzare gli aspetti chiave.
Per il progetto verrà utilizzato l'IDE Eclipse EE e alcuni suoi Plug-in che permettono di integrare tencologie come Maven e Gradle.
Nella parte finale della relazione verranno analizzati alcuni dati di test effettuati con Jmeter e VisualVM con lo scopo di monitorare il comportamento della macchina su cui si esegue l'applicazione web e monitorare il comportamento delle richieste che sopraggiungono al sistema dall'esterno.
Infine verranno svolte alcune considerazioni sul lavoro effettuato e le tecnologie utilizzate.

2.0 Servizio di Discovery

Il servizio di Discovery è il modo con cui le applicazioni e i (micro) servizi si localizzano a vicenda su una rete.
Tramite il service Discovery è possibile accedere ad una qualunque risorsa o servizio disponibile in rete.

Le implementazioni del Servizio di discovery icludono:

- Server centrale (o piu server) che mantengono gli indirizzi dei servizi
- Client che si connettono al server centrale per aggiornare e trovare indirizzi

Per effettuare una richiesta, l'applicazione deve conoscere la posizione di rete (indirizzo IP e porta) di un'istanza di servizio.
In un'applicazione tradizionale in esecuzione su hardware fisico, le posizioni di rete delle istanze di servizio sono relativamente statiche. Il codice può leggere per esempio le posizioni di rete da un file di configurazione che viene aggiornato occasionalmente.
In un'applicazione di microservizi moderna basata su cloud, tuttavia, questo è un problema molto più difficile da risolvere.
Le istanze del servizio cambiano posizione in maniera dinamica sulla rete. Conseguentemente il codice del Client ha bisogno di usare un meccanismo di Discovery piu elaborato per continuare a reperire correttamente il servizio.
Il servizio di Discovery (fig.1) si occupa proprio della reperibilità di questi servizi nel tempo e con condizioni non sempre favorevoli.

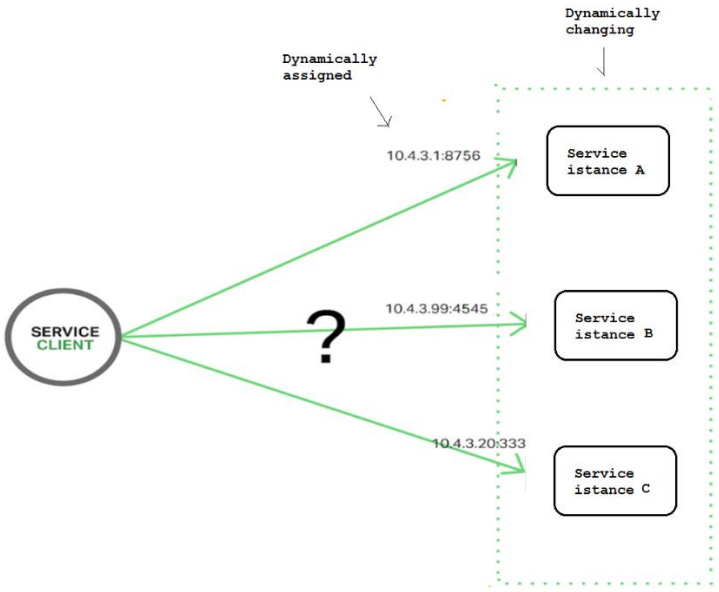


Fig.1 Service Discovery

2.1 Il modello del registro dei servizi

Il registro dei servizi è una parte fondamentale per il rilevamento dei servizi. È un database che contiene le informazioni sulle sue istanze e la loro posizione.
Le istanze di un servizio si registrano e deregistrano dinamicamente presso Il Service Registry e aggiornano costantemente le informazioni sontenuti nel registro in modo da essere utilizzate e consociute dagli altri servizi e/o istanze.
Di conseguenza un Service Registry deve essere sempre disponibile e costantemente aggiornato.
È possibile da parte dei CLient memorizzare in una cache le posizioni di rete ottenute dal registro del servizio ma tali informazioni diventano obsolete nel momento in cui la disponibilità del servizio di registro viene a mancare.
Il Service Registry puo controllare lo stato di vita di un istanza di un servizio, determinando se un particolare servizio riesce o no a gestire le richieste.
In basso viene riportato uno schema sul modello service registry (Fig.2):

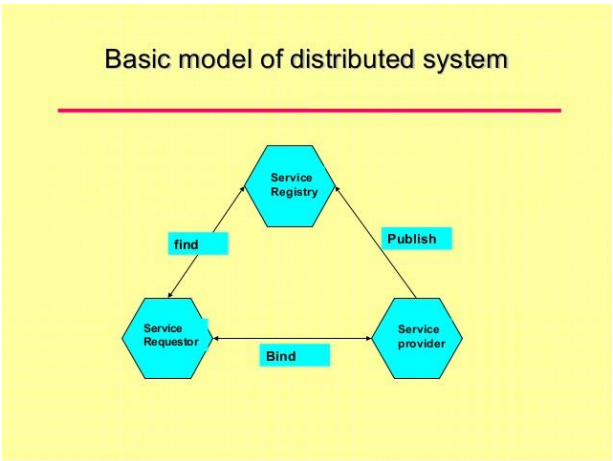


Fig.2 distribued system model [PRA15]

Le interazioni tra il registro e altri componenti possono essere divise in due gruppi, ciascuno con due sottogruppi:

- Interazioni tra microservizi e registro (registration)
    1. Self-registration
    2. Third-party registration

- Interazioni tra client e registro (discovery)
    1. Client-side Discovery
    2. Server-side Discovery

2.2 Interazione tra clienti e registro (Discovery)

Quando un cliente desidera accedere a un servizio, deve scoprire dove si trova il servizio e altre informazioni rilevanti per eseguire la richiesta.
Una moderna applicazione basata su microservizi viene generalmente eseguita in ambienti virtualizzati o containerizzati in cui il numero di istanze di un servizio e le relative posizioni cambiano in modo dinamico.

- Client Side Discovery:
  Quando viene fatta una richiesta a un servizio,, il client ottiene la locazione di un istanza di servizio interrogando un Service Registry, il quale conosce la posizione di tutte le altre istanze gli altri servizi

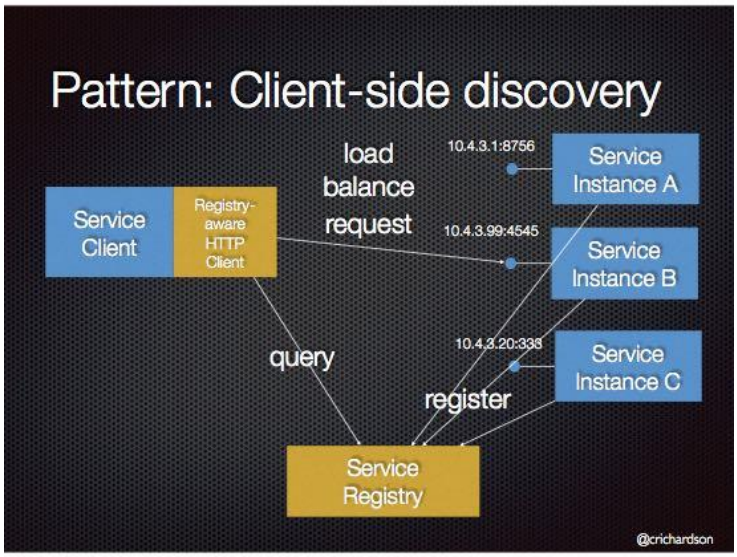


Fig.3 Client Side Discovery[CHR15]

- Server side discovery:
  Quando viene fatta una richiesta a un servizio, il Client fa una richiesta attraverso un router (load balancer) che si trova in una specifica posizione conosciuta.
  I router interroga il service Registry e avanza le richieste del client a una istanza di servizio disponibile.
  Il service Registry può anche essere incorporato nel router stesso.

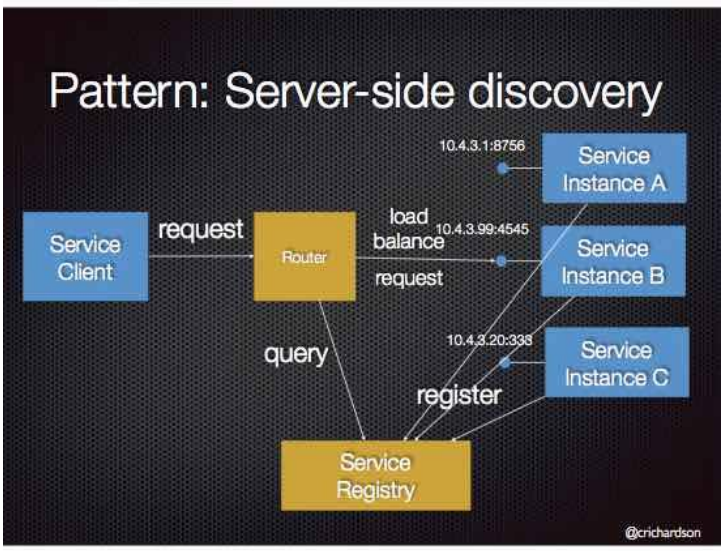


Fig.4 Server side Discovery [CHR15]

2.3 Interazione tra microservizi e il registro (registration)

Consideriamo uno scenario in cui è presente il modello Server-side Discovery o Client-side Discovery. Le istanze di servizio devono essere registrate con il service registry all'avvio in modo che possano essere scoperte e possano annullare la registrazioen all'arresto.

- Self-registraion:
  Un istanza di servizio è responsabile per registrare se stessa col service registry.
  All'avvio l'istanza si registra (host e indirizzo IP) con il service registry e si rende automaticamente localizzabile dagli altri servizi.
  Tipicamente un client puo rinnovare periodicamente la sua registrazione in questo modo il registro sa che il servizio è ancora attivo.
  Allo spegnimento della macchina per quel particolare servizio viene creata una richiesta in cui si avvisa il registro di tale azione.

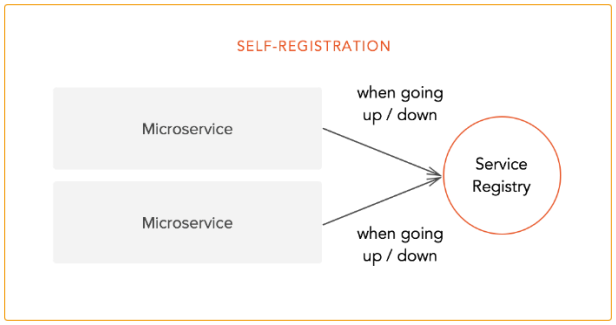


Fig.5 Self Registration [SEB15]

- Third party registration:
  Un'entità di terze parti è responsabile della registrazione e dell'annullamento della registrazione di un'istanza del servizio con il service registry. All'avvio, l'entità registra l'istanza del servizio con il service registry.

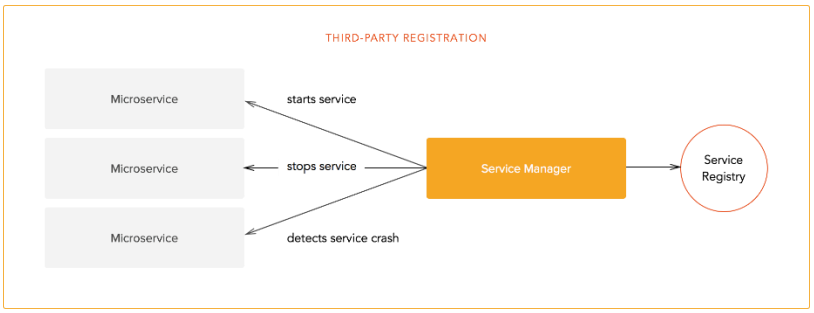


Fig.6 schema Third party registration[SEB15]

3.0 Bilanciamento del carico

Tecnica che consiste nel distribuire il carico di elaborazione di uno specifico servizio, ad esempio la fornitura di un sito web, tra più server, aumentando in questo modo scalabilità e affidabilità dell'architettura nel suo complesso
La scalabilità deriva dal fatto che, nel caso sia necessario, si possono aggiungere nuovi server al cluster, mentre la maggiore affidabilità deriva dal fatto che la rottura di uno dei server non compromette la fornitura del servizio (fault tolerance) agli utenti; non a caso i sistemi di load balancing in genere integrano dei sistemi di monitoraggio che escludono automaticamente dal cluster i server non raggiungibili ed evitano in questo modo di far fallire una porzione delle richieste di servizio degli utenti.

3.1 Server-Side

Può essere implementato in hardware o software. Il traffico viene inoltrato a un servizio dedicato che decide quale server contattare, utilizzando un algoritmo come round-robin, a una delle tante istanze. Questa modalità di funzionamento viene spesso definita "proxy" poiché il servizio dedicato funziona sia come bilanciamento del carico che come proxy inverso. Il vantaggio principale è la semplicità. Il meccanismo di bilanciamento del carico e del servizio è in genere incorporato nel contenitore e non è necessario preoccuparsi dell'installazione o della gestione di tali componenti. Inoltre, il client (ad es. Il nostro servizio) non deve essere a conoscenza del registro del servizio - il servizio di bilanciamento del carico si occupa di questo per noi. Essere dipendenti dal bilanciamento del carico per instradare tutte le chiamate diminuisce la resilienza e il bilanciatore del carico potrebbe teoricamente diventare un collo di bottiglia.

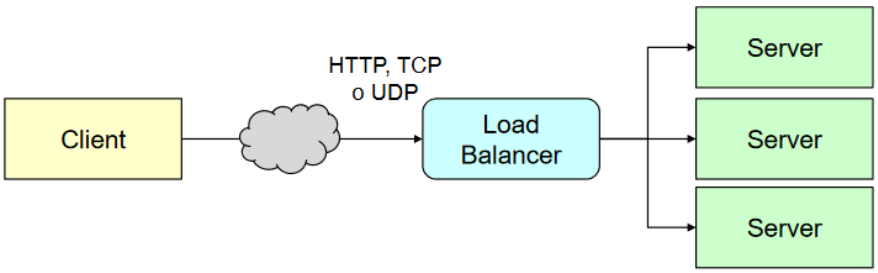


Fig.7 Server side Load balancer [EBE15]

3.2 Client-Side

Invece di affidarsi a un altro servizio per distribuire il carico, il client stesso è responsabile per decidere dove inviare il traffico anche utilizzando un algoritmo come round-robin. Può scoprire le istanze,

tramite l'individuazione dei servizi (Eureka), oppure può essere configurato con un elenco predefinito statico.
In questo modello spetta al cliente interrogare un servizio di discovery per ottenere informazioni sugli indirizzi effettivi (IP, nomi host, porte) dei servizi che si deve chiamare.
Inoltre, per non dover interrogare il servizio di discovery per ciascuna chiamata in arrivo, ogni client mantiene in genere una cache locale di endpoint che deve essere mantenuta aggiornata con le informazioni principali del servizio di discovery.
Alcuni vantaggi del bilanciamento del carico lato client sono la resilienza, il decentramento e nessun collo di bottiglia centrale dal momento che ciascun consumatore di servizi mantiene il proprio registro degli endpoint del produttore. Alcuni inconvenienti sono la maggiore complessità del servizio interno e il rischio egistri locali contenenti voci obsolete.

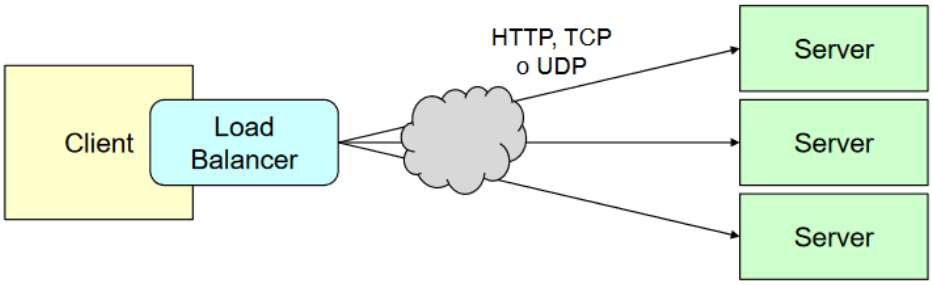


Fig.8 Client side Load Balancer [EBE15]

4.0 Netflix Api

Spring Cloud è un progetto basato su Spring Boot che fornisce degli strumenti per affrontare alcune problematiche comuni nello sviluppo delle applicazioni distribuite(applicazioni a miscroservizi) sulla base di un insieme di pattern:

- I servizi e le applicazioni Spring Cloud possono essere rilasciati in ogni ambiente distribuito (ambienti fisici, virtuali e Cloud ma anche sul proprio personal computer.
- Cosi come Spring è composto da altri sottoprogetti tra cui Spring Framework, Spring Boot e Spring Cloud, anche Spring Cloud a sua volta è composto da altri progetti.

Spring Cloud fornisce delle librerie per applicare alcuni pattern comuni utili nello sviluppo di applicazioni disctribuite e a miscroservizi tra cui:

- gestione delle configurazioni (centralizzata, con controllo delle versioni)
- registrazioni e discovery di servizi con Eureka
- Invocazioni dei servizi
- Bilanciamento del carico e Routing dinamico con Ribbon
- Circuit breaker con Hystrix
- Monitoraggio con Hystrix dashboard e Turbine
- API gateway e routing con Zuul
- OAuth 2.0 con Spring Cloud + Spring Security OAuth2

I progetti Spring Cloud sono tutti basati su Spring Boot che semplifica lo sviluppo e la gestione delle applicazioni, va comunque osservato che Spring Cloud è basato su un processo di sturtup (per la definizione dell'application context) modificato rispetto a quello di Spring Boot.
In quest documento il significato dei termini Client Server è relativo e viene usato per descrivere il ruolo svolto da due elementi nel contesto di una specifica relazione/interazione.

Per esempio un microservizio è generalmente un server (dal punto di vista applicativo) , ma nei confronti di altri servzi applicaztivi e di utilità (come il servizio di discovery) potrebbe anche comportarsi da client

5.0 Servizio di Discovery con Eureka

Eureka è un progetto impementato e attualmente utilizzato da Netflix OSS rilasciato con licenza opensource basato su REST (Representational State Transfer) utilizzato principalmente nel cloud AWS(amazon web service) per l'individuazione dei servizi ai fini del bilanciamento del carico e del failover server di livello intermedio. Eureka oltre ad avere una compnente Server. Eureka ha anche un componente client basato su Java, il client Eureka, che semplifica notevolmente le interazioni con il servizio. Il client ha anche un sistema di bilanciamento del carico integrato che esegue il bilanciamento del carico di base round robin

L'architettura Netflix Eureka è costituita da due componenti, Server e Client.
Il server è un'applicazione standalone ed è responsabile di:

- gestisce il registro delle istanze di servizio,
- fornire mezzi per registrare, de-registrare e richiedere istanze con il Registro di sistema,
- propagazione del registro ad altre istanze Eureka (Server o Clienti).

Il cliente fa parte dell' istanza del servizio e ha responsabilità di:

- registrare e annullare la registrazione di un istanza di servizio con Eureka Server,
- mantenere viva la connessione con Eureka Server,
- recuperare e memorizzare le informazioni di scoperta dalla Eureka Server.

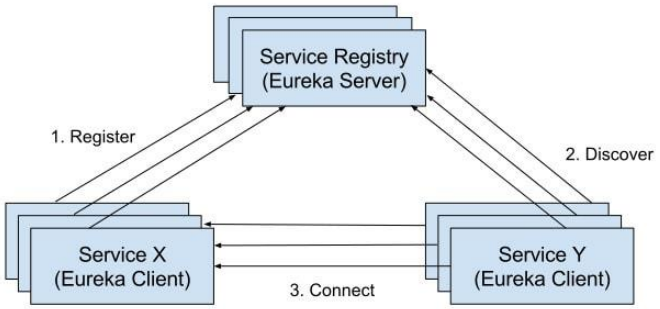


Fig.9 Service Registry [ORL16]

Eureka 2.0

È un evoluzione dell'originale versione 1.0, che è attualmente in sviluppo il quale cerca di essere molto piu scalabile.
Il sistema Eureka stesso consiste in cluster di scrittura e cluster di lettura.
Il cluster di scrittura è un sottosistema senza stato (stateful), il quale gestisce le registrazioni dei client e mantiene un registro interno.
Il contenuto dei registri dei cluster di scrittura vengono letti dai cluster di lettura che a sua volta vengono usati dai client Eureka .
Poiché il cluster di lettura è effettivamente un livello di cache, può essere facilmente e rapidamente scalato a seconda del volume del traffico.

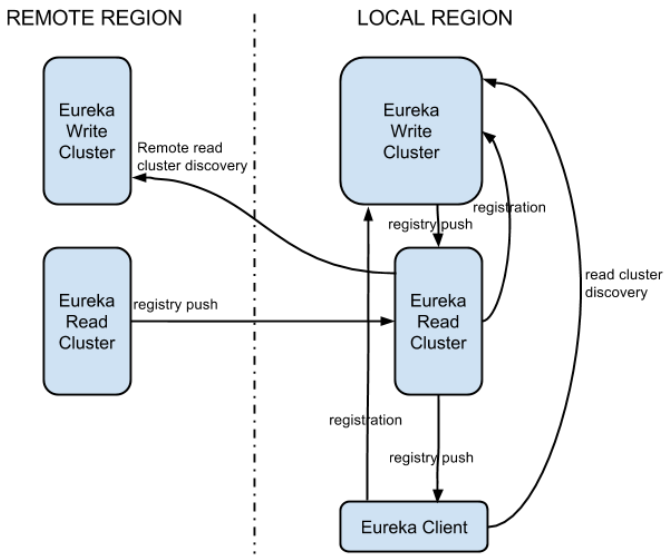


Fig.10 [DAV12]

5.1 Eureka Client

Per poter essere scoperti i Client Eureka devono preventivamente

- Registrarsi inviando la richiesta al Server Eureka
- mantenere le informazioni del registro del Server Eureka aggiornate e inviare segnalazioni sul proprio stato di salute
- segnalare l'uscita di quel nodo annullando la registrazione.

Un singolo Client puo registrare diverse istanze di servizio
Ogni registrazione viene gestita tramite una connessione separata al server di scrittura.
Una volta completata la registrazione, dopo ogni 30 secondi, il client Eureka invia un segnale (heartbeat) al server Eureka per notificare lo stato.
Dal momento che il Client viene registrato avrà la possibilitò di effettuare le richieste di aggornamento al server cambiando i dati della propria istanza.
Se dopo una certa quantità di tempo il server Eureka non riceve alcun heartbeat dal client Eureka, annulla la registrazione dell'istanza del client dal registro del servizio e invia il registro aggiornato a tutti i peer e ai client Eureka.
I client che volgiono annullare la registrazione presso il registro dovrebbero notificare la loro uscita al server di registrazione prima di disconnettersi .

5.2 Individuazione del registro

I Client una volta scoperto la posizione del servizio dal server di scrittura, puo successivamente registrarsi a più istanze e aggiornare le proprie informazioni sul registro attraverso i server di lettura.
I server Eureka responsabili della gestione di queste azioni formano il cluster di lettura.

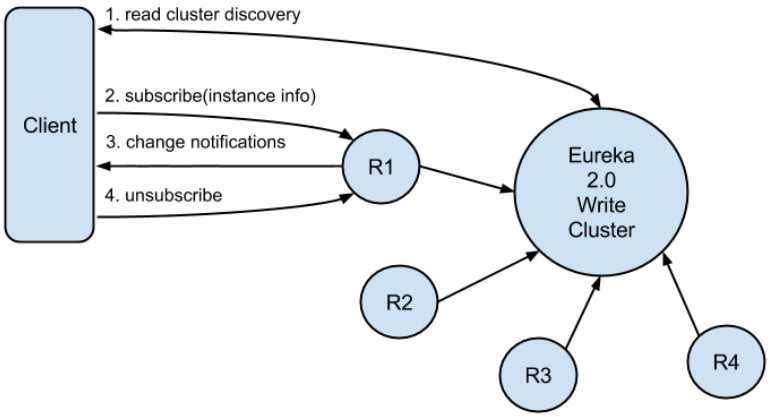


Fig.11 [DAV12]

5.3 Modello del Servizio di registro

Eureka è stato progettato per funzionare con diversi fornitori di cloud e data center quindi il suo modello dei dati è basato sull'estesibilità per potersi adattare alle implementazioni attualei e future. È possibile utilizzare modelli differenti dei dati, per esempio utilizzare un modello base o un modello basato su Amazon Web Service o VPC.

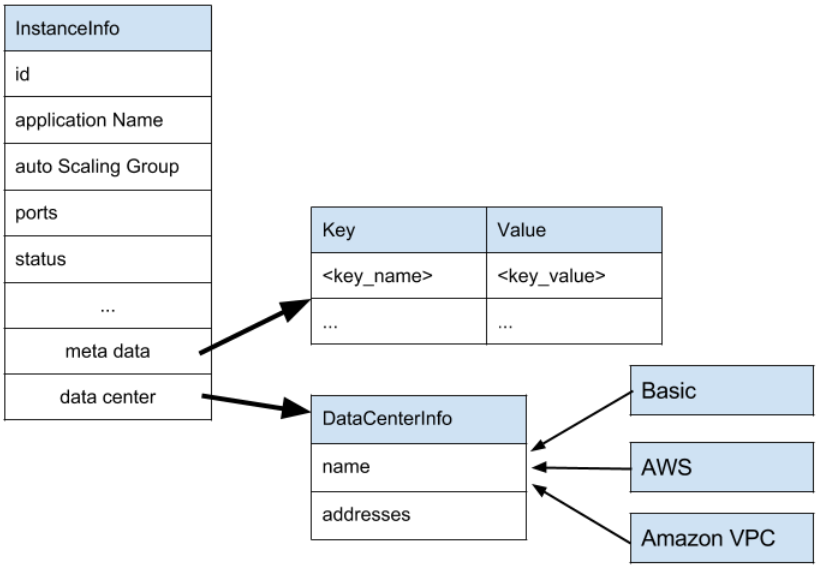


Fig.12 modello registro [DAV12]

L'insieme degli attributi predefiniti delle istanze del servizio possono essere estesi tramite un set personalizzato di coppie chiave / valore che possono essere aggiunte nei metadati.
La topologia di rete non è predefinita e viene concretizzata da estensioni specifiche del cloud. Per la distribuzione in AWS viene quindi fornito un semplice modello IP pubblico / privato, ma per VPC è supportato il supporto multi NIC / ENI.

5.4 Modello di registrazione con Interest

Un modello di registrazione consiste in un insieme predefinito di classi atomiche "Interest":

- applicaztion interest: tutte le istanze di servizio appartenenti a una determinata applicazione
- Vip Interest:tutte le istanze di servizio apaprtengono a un particolare indirizzo virtuale Eureka (VIP)
- Istance id: specifica istanza con un determinato id
- Full registry: tipo di interest speciale che denota tutte le voci nel registro, (dovrebbe essere usato raramente in quanto potrebbe generare un traffico enorme per i registri di grandi dimensioni)

Gli Interest possono essere combinati in strutture composite con operatori OR:
Ad esempio, supponiamo di essere interessati alle applicazioni "eureka_write.*", "eureka_read" e "atlas".
Possiamo registrarci a piu applicazioni attraverso i seguenti Interest:

application("eureka.", Operator.Like) OR application("atlas", Operator.Equals)*

le quali tradotte usando il Client Eureka diventa:

```
Interests.forSome(
             Interests.forApplications(Operator.Like, "eureka.*"),
             Interests.forApplications(Operator.Equals, "atlas")
             );
```

5.5 Dashboard

Eureka 2.0 Dashboard è un componente opzionale, per la gestione e il monitoraggio generale dei cluster Eureka. Fornisce una visualizzazione a livello di cluster, con strumenti per eseguire il drill-down su un'istanza specifica per una facile risoluzione dei problemi o diagnosi del sistema.

Questa applicazione fornisce una semplice interfaccia grafica per amministrare l'infrastruttura delle applicazioni Spring Cloud. È un fork di Spring Boot Admin per gestire le applicazioni registrate nel registro di servizio (Netflix Eureka e AWS Beanstalk).

Al momento fornisce le seguenti funzionalità per ogni applicazione registrata (la maggior parte poi ereditata da spring-boot-admin).

- mostra in nome/id e il numero di versione
- mostra lo stato di salute
- Mostra le proprietà del sistema Java, dell'ambiente Java e di Spring
- Valutatori della memoria e della JVM(Java Virtual machine)
- Valutatore dei Contatori
- Valutatori delle sorgenti dei dati
- Gestione facilitata dei logger
- Interazione con i bean-JMX
- Mostra informazioni sullo stato di ogni thread
- Mostra la cronologia delle registrazioni delle applicazioni (da parte del Server Eureka)

È possibile configurare facilmente il Dashboard attraverso l'aggiunta delle dipendenze nel file pom.xml nel caso si utilizzi Maven o build.gradle altrimenti.

```
<dependency>
```

```
    <groupId>com.github.vanroy</groupId>
    <artifactId>spring-cloud-dashboard</artifactId>
    <version>1.2.0.RELEASE</version>
</dependency>
```

Attraverso una singola annotazione è possibile creade un istanza del Dashboard

```
@SpringBootApplication
@EnableEurekaServer
@EnableCloudDashboard
@EnableDiscoveryClient
public class EurekaServerApplication {

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

Spring Cloud Dashboard usa Apache HTTP Client per interrogare gli endpoint dell'istanza.
A volte è possibile che questi endpoint siano protetti. Sono disponibili configurazioni per personalizzare il client http con le intestazioni di pre-autorizzazione di base.

```
spring:
  cloud:
    dashboard:
      http:
        # Credenziali di base
        username: user
        password: password

        # Valori di Default opzionali
        maxConnection: 100
          connectTimeout: 1000
          socketTimeout: 2000
          requestTimeout: 1000
```

Digitando su browser è possibile visualizzare l'interfaccia:

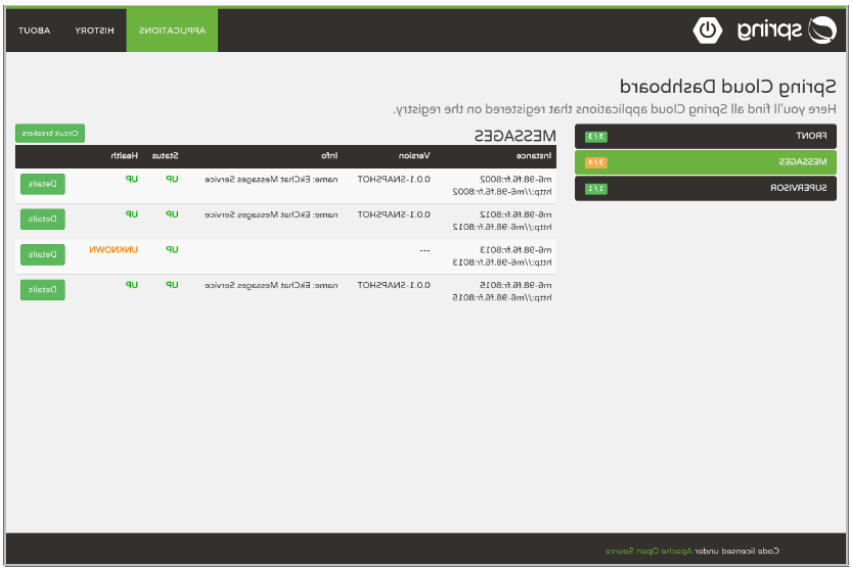


Fig.13 Dashboard

5.6 Teorema CAP

In informatica teorica, il teorema CAP, noto anche come teorema di Brewer, afferma che è impossibile per un sistema informatico distribuito fornire simultaneamente tutte e tre le seguenti garanzie:

- *Coerenza* : (tutti i nodi vedano gli stessi dati nello stesso momento)se viene scritto un dato in un nodo e viene letto da un altro nodo in un sistema distribuito, il sistema ritornerà l'ultimo valore scritto
- *Disponibilità* : (la garanzia che ogni richiesta riceva una risposta su ciò che è riuscito o fallito)Ogni nodo di un sistema distribuito deve sempre rispondere ad una query a meno che non sia indisponibile.
- *Tolleranza di partizione* (il sistema continua a funzionare nonostante arbitrarie perdite di messaggi)è la capacità di un sistema di essere tollerante ad una aggiunta o una rimozione di un nodo nel sistema distribuito (*partizionamento*) o alla perdita di messaggi sulla rete.

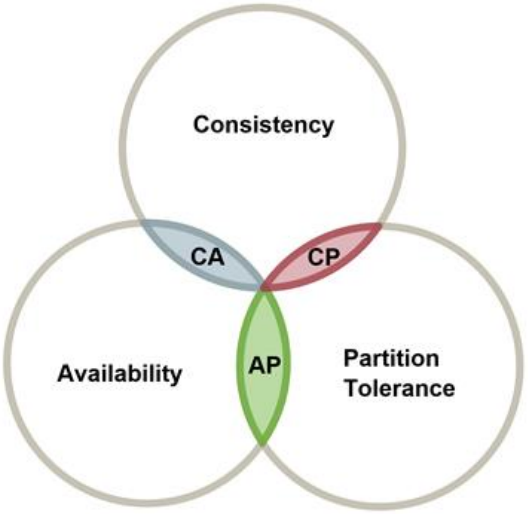


Fig.14 CAP Theorem [BRE00[

Secondo il teorema, un sistema distribuito è in grado di soddisfare al massimo due di queste garanzie allo stesso tempo, ma non tutte e tre.

Paragonandolo con un esempio si consideri un sistema distribuito e si supponga di aggiornare un dato su un *nodo 1* e di leggerlo da un *nodo 2*, si verificheranno le seguenti conseguenze:

1. Il *nodo 2* deve ritornare l'ultima "miglior" versione del dato (quella consistente) per non violare il principio della *Consistenza*
2. Si potrebbe attendere la propagazione del dato modificato nel *nodo 2* e, quest'ultimo, potrebbe mettersi in attesa della versione più aggiornata, ma, in un sistema distribuito, si ha un'alta possibilità di perdita del messaggio di aggiornamento e il *nodo 2* potrebbe attenderlo a lungo. Così non risponderebbe alle query (*indisponibilità*), violando il principio dell'*Availability*
3. Se volessimo garantire sia la *Consistency* che l'*Availability*, non dovremmo partizionare la rete, violando il principio del *Partition-Tolerance*

Dal punto di vista di questo teorema, il cluster di scrittura di Eureka è un sistema AP (disponibilità e tolleranza di partizione). Questa scelta è guidata dai principali requisiti dei servizi di Discovery basati su cloud.
Nel cloud, in particolare per le distribuzioni di grandi dimensioni, si possono sempre verificare degli errori e questo potrebbe comportare in questo caso, un fallimento del Server Eureka, del Client registrato o della partizione della rete.
In queste circostanze, Eureka rimane disponibile continuando a fornire informazioni del registro e accettando nuove registrazioni da ciascun nodo disponibile in modo separato ma dato che Eureka sceglie di mantenere la disponibilità, i dati potrebbero non sono essere coerenti tra questi nodi.
Dato che questo modello causa sempre un certo livello di pesantezza dei dati del registro, dovrebbe essere integrato da un giusto bilanciamento del carico lato client e da meccanismi di failover.
Nell'ecosistema Netflix queste funzionalità sono fornite da Ribbon.

5.7 Tipi di Client

Eureka provvede delle interfacce per due differenti tipi di Client specifici “interest discovery” e “istance discovery”
Un prerequisito per la creazione di qualsiasi client eureka è un ServerResolver che specifica il server eureka remoto a cui connettersi. Eureka 2.0 include alcuni metodi standard per la creazione di server resolver con `ServerResolvers`. Vedi la classe javadoc per le specifiche di utilizzo.

Interest Client

L’interestclient è usato per leggere le informazioni sulle istanze registrate da un server Eureka remoto basati su specifici creiteri di Interest.

```
EurekaInterestClient registrationClient = new EurekaInterestClientBuilder()
.withServerResolver(myResolver)
.build();
```

CONFIGURAZIONE

L’istanza `EurekaInterestClientBuilder` possono essere fornite facoltativamente le seguenti classi di configurazioni.

- `EurekaTransportConfig`: classe di configurazione che gestisce le configurazioni a livello di trasporto.
- `EurekaRegistryConfig`: cgestisce le configurazion delle registrazioni
- `EurekaClientMetricFactory`: cclasse per la valutazione a livello di client.
- `EurekaRegistryMetricFactory`: cvalutatore per metriche generate dal registro Eureka lato client, è disponibile un implementazione Noop.

Utilizzo dell’interest Client

Come prima cose vengono decisi i criteri dell’Interest lper le istanze che noi vogliamo leggere da dei server Eureka remoti.
Di default Eureka supporta nterests a livello di applicazione, a livello di Vip address e a livello di istanza.

`Interest<InstanceInfo> myInterest = Interests.forApplications("WriteServer", "ReadServer");`

Per poter generare uno stream di notifiche riguardo al cambiamento delle informazioni dell'istanza si può chiamare il metodo `interestClient.forInterest(myInterest)`.

Questa funzione ritorna un rx.Observable di `ChangeNotification<InstanceInfo>` che puo essere registrato e consumato.

```
interestClient.forInterest(myInterest).subscribe(
            new Subscriber<ChangeNotification<InstanceInfo>>() {
                @Override
                public void onCompleted() {
                    System.out.println("Change notification stream closed");
                }

                @Override
                public void onError(Throwable e) {
                    System.out.println("Error in the notification channel: " + e);
                }

                @Override
                public void onNext(ChangeNotification<InstanceInfo>
changeNotification) {
                    System.out.println("Received notification: " +
changeNotification);
                }
            });
```

È possibile utilizzare la funzione Interest passando un buffer e una lista di istantanee attraverso il metodo .compose.

I metodi ereditati dalla classe c permetteranno di avvisarci nel caso la registrazione sia avvenuta con successo o in caso di errori nell'inoltrare la notifica.

```
   interestClient.forInterest(forFullRegistry())
    .compose(ChangeNotificationFunctions.<InstanceInfo>buffers())
    .compose(ChangeNotificationFunctions.<InstanceInfo>snapshots())
    .subscribe(new Subscriber<LinkedHashSet<InstanceInfo>>() {
        @Override
        public void onCompleted() {
            System.out.println("Change notification stream closed");
        }

        @Override
        public void onError(Throwable e) {
            System.out.println("Error in the notification channel: " + e);
        }

        @Override
        public void onNext(LinkedHashSet<InstanceInfo> instanceInfos) {
            System.out.println("Received new snapshot: " + instanceInfos);
        }
    });
```

5.8 Registration Client

Il registration Client è usato per registrare/aggiornare e annullare informazioni sull'istanza con un server Eureka remoto, in questo modo p possibile essere scoperto da altri interest client.

Il client di registrazione può essere utilizzato per mantenere le informazioni di istanza per più entità registrate.

```
EurekaRegistrationClient registrationClient = new EurekaRegistrationClientBuilder()
            .withServerResolver(myResolver)
            .build();
```

Configurazione Registration Client

Come prerequisito per la registrazione bisogna creare un modello `InstanceInfo(s)`.
Per ogni registrazione, il client richiede che l'istanza IstanceInfo sia inviata come un oggetto `rx.Observable<InstanceInfo>`.
Successivamente si crea la variabile `staticInstanceInfo` nacessaria a catturare la risposta di tipo `RegistrationObservable` nella variabile registrationObservable, che verrà poi sottoscritta con il metodo subscribe().
Il processo di registrazione inizia proprio quando questa RegistrationObservable viene sottoscritta da parte del Server.
Il RegistratonObservable espone ache un metodo per vedere i risultati iniziali di una registrazione tramite la funzione onComplited() che viene attivata in caso di corretta sottoscrizione.
Una voltaottenuta la sottoscrizione è possibile poi implementare la logica che si preferisce e aggiornare il registro tramite la creazione e l'invio delle informazioni tramite la funzione onNext(IstanceInfo) della variabile di tipo BehaviorSubject<InstanceInfo>.
Una volta che si desidera di annullare la sottoscrizione verrà richiamato il metodo unsubscribe() dell'istanza Subscription.

Esempio registrazione iniziale

```
Observable<InstanceInfo> staticInstanceInfo = Observable.just(myInstanceInfo);
    RegistrationObservable registrationObservable =
registrationClient.register(staticInstanceInfo);
    registrationObservable.subscribe();
    registrationObservable.initialRegistrationResult().doOnCompleted(new Action0() {
        @Override
        public void call() {
            System.out.println("Initial registration completed");
        }
    }).subscribe();
```

esempio aggiornamento e annullamento sottoscrizione.

```
   BehaviorSubject<InstanceInfo> infoSubject = BehaviorSubject.create();
    Subscription subscription =
registrationClient.register(infoSubject).subscribe();
    // initial registration
    infoSubject.onNext(myInstanceInfo);

    // some business logic that changed instanceInfo
    // update instanceInfo
    InstanceInfo updatedInfo = new
Builder().withInstanceInfo(myInstanceInfo).withStatus(Status.DOWN).build();
    infoSubject.onNext(updatedInfo);

    // unregistering
    subscription.unsubscribe();
```

5.9 Configurazione

Per impostazione predefinita, BasicEurekaTransportConfig e BasicEurekaTransportConfig vengono utilizzati per fornire il caricamento della configurazione dalle proprietà di sistema.

EUREKA TRANSPORTCONFIG

Le seguenti proprietà possono essere cambiate:

- heartbeatIntervalMs- regola l'intervallo con cui gli heartbeat vengono inviati sui canali di trasporto. Predefinito a 30 secondi.
- connectionAutoTimeoutMs: tutte le connessioni di trasporto di eureka scadranno dopo un periodo di (connectionAutoTimeoutMs / 2, 1 + connectionAutoTimeoutMs / 2) e si ristabiliranno automaticamente. Impostare su 0 per disabilitare questa funzione.
- Codec: protocollo di codifica a livello di trasporto. Avro e Json sono disponibili.

EUREKAREGISTRYCONFIG

Proprietà modificabili:

- evictionTimeoutMs: quantità di tempo da aspettare per un IstanceInfo sconosciuta per essere rimossa.
- EvictionStrategyType:la strategia per la rimozione delle istanze, attualmente è disponibile solo il valore predefinito di PercentageDrop.
- evictionStrategyValue:

5.10 Server Eureka

Eureka usa due differenti tipi di server di default, server di scrittura e server di lettura.
È disponibile un modulo “ponte” opzionale che può essere configurato come componente di un server di scrittura che consente la migrazione di dati da un sistema eureka 1.0 attualmente in uso al sistema 2.0.

Configurazioni Comuni lto Server
La combinazione di resolverType e serverType definisce il cluster dei server di scrittura

- il ResolverType puo essere fixed(legge direttamente dalla lista dei server) o DNS(il nome della lista dei Server puo essere presa cone entry dns o estesa)
- ServerList è un elenco delimitato da virgole con ogni voce nel formato “name:registrationPort:interestPort:replicationPort

esempio:

```
eureka.common.writeCluster.resolverType=fixed
eureka.common.writeCluster.serverList=["localhost:12102:12103:12104","localho
st:12112:12113:12114"]
```

Questa configurazione comune per identificare tutti i server di scrittura è necessaria per tutti i tipi di server.

Ogni server avrà anche bisogno di una configurazione per autoidentificarsi.

- appName- nome dell'applicazione server nel suo IstanceInfo.
- vipAddress-
- dataCenterInfo.type-Basic o AWS

esempio :

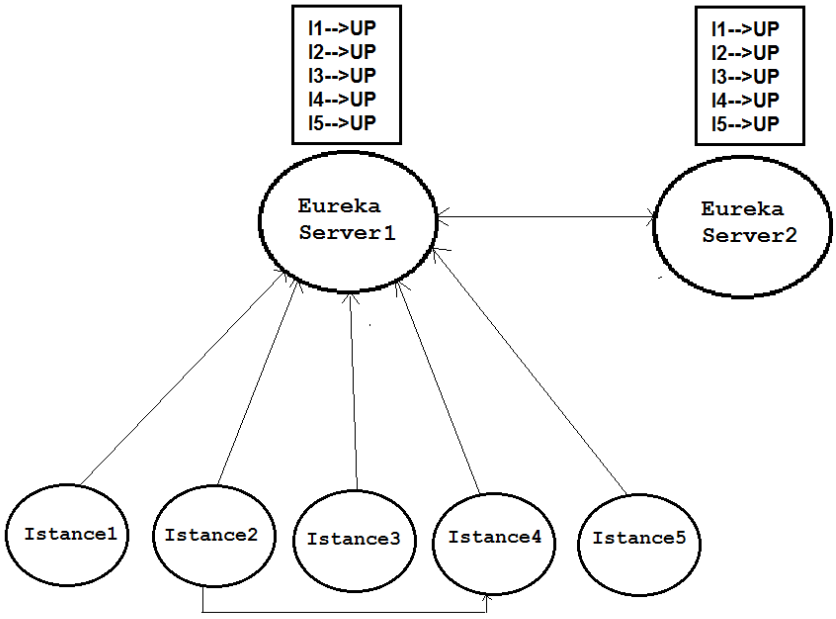
```
eureka.instanceInfo.appName=eureka-write-cluster
eureka.instanceInfo.vipAddress=eureka-write-cluster
eureka.dataCenterInfo.type=Basic
```

Earthbeats

Il Client e il Server Eureka implementano in protocollo chiamato "heartbeat"ovvero un Client deve inviare regolarmente segnali heartbeat al Server. Il server si aspetta questi segnali in modo da mantenere l'istanza nel registro e di aggiornare le sue informaioni, altrimenti l'istanza viene rimossa dal registro.
Il frammento di tempo è configurabile e il segnale heartbeats puo anche specificare lo stato dell'istanza del servizio: UP,DOWN,OUT_OF_SERVICE

5.11 Self-preservation

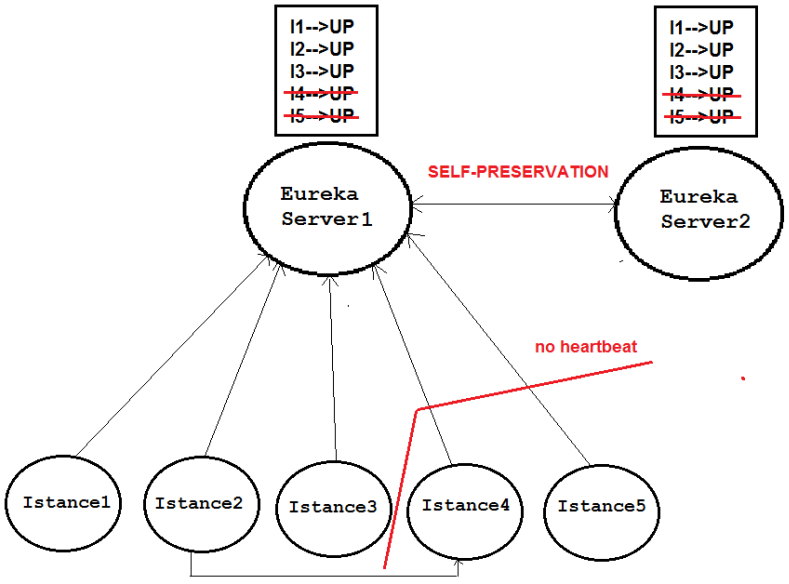
Eureka Server ha una funzione di protezione: nel caso in cui un determinato numero di Istanze non riesca a inviare heartbeat in un determinato intervallo di tempo, il Server non le rimuoverà dal registro. Ritiene che si sia verificata una partizione di rete e attenderà il ritorno di queste istanze. Questa funzione è molto utile nei deployment Cloud e può essere disattivata per i servizi collocati in un data center privato.



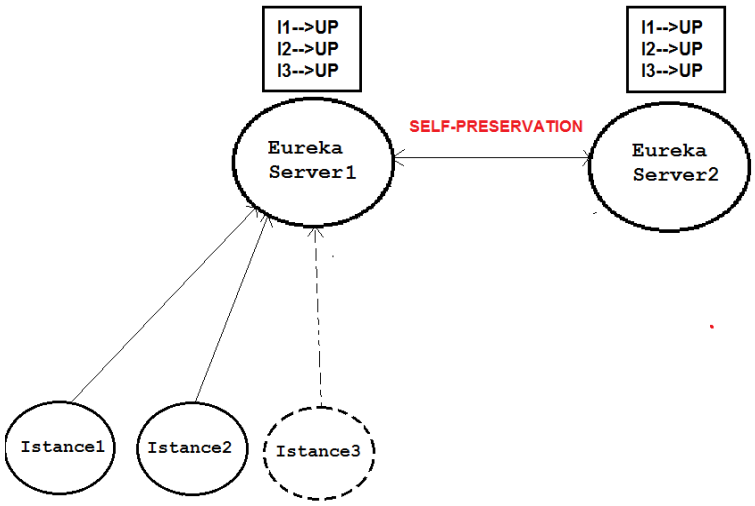
Supponiamo che tutti i microservizi sono in stato di salute buono e registrati con EurekaServer1.
Le istanze inviano correttamente heartbeat al server1.
Le istanze registrano e inviano heartbeats solo al primo server configurato nella lista service-url .

```
eureka.client.service-url.defaultZone=server1,server2
```

I Server Eureka replicano le informazioni del registro con i peer adiacenti e il registro indica che tutte le istanze sono in stato UP.
Suppponiamo anche che l'istanza 2 ha usato invocare l'istanza 4 dopo averla scoperta dal registro di Eureka Server.



Quando si verificherà una partizione della rete potremmo avere che l'istanza 4 e 5 perdano la connettività coi server, in ogni modo l'istanza 2 continua ad avere connettività con l'istanza 4.
Eureka server allora elimina l'istanza 4 e 5 dal registro da quando non riceve piu heartbeats.
Quindi inizierà ad osservare che improvvisamente ha perso più del 15% degli heartbeat e entrerà nella modalità self-preservation.



Da adesso in poi Eureka smetterà di eliminare le istanze dal registro anche se le istanze rimanenti smetteranno di funzionare.

Motivazione della Self-Preservation

La modalità self-preservation può essere giustificata per due motivi.

- I server che non ricevono gli heartbeats potrrebbero avere una scarsa connettività di rete(non vuol necessariamente dire che i client sono giu) la quale puo essere risolta presto.
- Anche se la connessione è persa tra i server e più client, i client possono mantenere la connettività con le altre con cui è collegata.

CONFIGURAZIONE
Le configurazioni che possono impattare direttamente o indirettamente l'ambiente della modalità self-preservation.

"lease-renewal-interval-in-seconds "indica la frequenza con cui il client puo mandare heartbeats al server per far sapere che quello è ancora "vivo".
Self-preservation assume che gli heartbeats siano ricevuti in un intervallo di 30 secondi di Default.

```
eureka.instance.lease-renewal-interval-in-seconds = 30
```

"lease-expiration-duration-in-seconds "Indica il tempo che il server aspetta da quando ha ricevuto l'ultimo heartbeat prima di eliminare una voce dal registro.
Questo valore dovrebbe essere piu grande di `lease-renewal-interval-in-seconds`
Il valore di quesstto parametro è importante perché impattera sulla salute del registro stesso.
Settando questo valore troppo basso potrebbe rendere il sistema intollerabile alle perdite brevi di connessione.

```
ureka.instance.lease-expiration-duration-in-seconds = 90
```

Viene eseguito uno scheduler a questa frequenza il quale alimina le istanze dal registro se i ltempo ha superato la soglia `lease-expiration-duration-in-seconds`.
Settando questo valore troppo alto si ritarderà il tempo per entrare in modalità self-preservation .

```
eureka.server.eviction-interval-timer-in-ms = 60 * 1000
```

Valore usato per calcolare gli heartbeat attesi al minuto (indica che ha una tollerabilità di perdita del 15%

```
eureka.server.renewal-percent-threshold = 0.85
```

Viene eseguito unoscheduler a questa frequenza attraverso il quale calcola gli heartbeat attesi al minuto .

```
eureka.server.renewal-threshold-update-interval-ms = 15 * 60 * 1000
```

La modalità Self-Preservation puo essere disattivata se necessario.

```
eureka.server.enable-self-preservation = true
```

CALCOLO
I Server Eureka entra in modalità Sef-preservation se il numero attuale degli heartbeats nell'ultimo minuto è inferiore al numero dei heartbeats atteso al minuto.

NUMERO DEGLI HERATBEATS ATTESI AL MNUTO
Il codice Netflix assume che questi heartbeats siano sempre ricevuti a un intervallo di 30 secondi per questocalcolo.
Supponiamo che il numero delle istanze applicative registrate nel Server di regstrazione a un certo punto siano N e il parametro "renewal-percent-threshold sia 0.85".

- numero delgi heartbeats atteso da un istanza / minuto=2
- numero delgi heartbeat attesi da N istanze / minuto=2*N
- minimi heratbeat attesi / minuto= 2*N*0,85

Anche se N è variabile , 2*N*0.85 viene calcolato ogni 15 minuti di Default (o basato sul parametro `renewal-threshold-update-interval-ms`)

NUMERO EFFETTIVO DEGLI HEARTBEAT AL MINUTO
Questo è calcolato con uno scheduler che si avvia con una frequeenza di 1 minuto.ù
Inoltre, come descritto sopra, due scheduler funzionano in modo indipendente per calcolare il numero effettivo e previsto di heartbeat.
Tuttavia è un altro schedulatore che confronta questi due valori e identifica se il sistema è in modalità di autoconservazione - che è `EvictionTask`.
Questo cheduler si avvia con una frequenza di `eviction-interval-timer-in-ms` ed elimina le istanze scadute, tuttavia controlla se il sistema ha raggiunto la modalità di self-preservation priman di eliminare l'istanza dal registro.

5.12 Esempio di applicazione

In questo esempio implementeremo un servizio di registrazione Eureka e un client eureka che dovrà prima registrarsi presso il server di registrazione e successivamente interrogare tale servizio per sapere il nome e la porta del servizio.

Lato Server

Come prima cosa è necessario aver creato un progetto Spring Boot, è possibile utilizzare sia Maven che Gradle.
Spring mette a disposizione una web form grazie al quale è possibile creare un progetto Spring Boot da zero in pochi secondi.

Attraverso questa form è possibile specificare se utilizzare Maven o Gradle, la versione di Spring Boot e una serie di metadata e dipendenze iniziali.
In questo esempio è stato creato un progetto Spring Boot 1.5.3 attraverso Gradle.
Dopo aver importato il progetto in Eclipse con il plugin GradleSTS, sarà necessario andare a modificare il file Build.gradle e inserire le repository e dipendenze necessarie a utilizzare Maven e le Api Spring Cloud.

```
buildscript {
    ext {
          springBootVersion = '1.5.2.RELEASE'
    }
    repositories {
          mavenCentral()
    }
    dependencies {
          classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'eureka-service'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencyManagement {
    imports {
          mavenBom 'org.springframework.cloud:spring-cloud-dependencies:Camden.SR5'
    }
}

dependencies {
```

```
    compile('org.springframework.cloud:spring-cloud-starter-eureka-server')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}


eclipse {
    classpath {
            containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
            containers
'org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.internal.debug.ui.launcher.
StandardVMType/JavaSE-1.8'
    }
    }
```

- Buildscript:questo blocco viene valutato rima di qualsiasi altro blocco nello script(indipendentemente dalla posizione in cui sis trova nel file).Determina quali plugin, classi di Task, e altre classi sono disponibili per essere usate nel resto del buildscript.Se si vogliono utilizzare plugin di terze parti, o classi di task o altre classi bosogna specificare le corrispettive dipendenze in questo blocco.
- **apply** : applica i plugin specificati
- `jar: il task jar genera un jar contenente le classi e le risorse del progetto.Viene messo nella directory "build/libs" e viene chiamato "NomeProgetto-versione.jar".`
- **sourceCompatibility:** La compatibilità della versione Java da utilizzare durante la compilazione del codice sorgente.
- **Targhet compatibility:**La versione java per la generazione delle classi
- **repositories:**`utilizza il repository Maven per attingere dalle librerie, indica che devono essere risolte le dipendenze dalla centrale Maven.`
- dependencyManagement
- **dependencies:Viene specificata una dipendenza a Spring Cloud Eureka**

Successivamente dovremo modificare il file principale java dove viene lanciata l'applicazione, per abilitare Spring boot ad essere riconosciuto come Server di Registrazione Aureka sarà sufficiente specificare la notazione @EnableEurekaServer sopra la classe prima della notazione @SpringBootApplication.

```
@EnableEurekaServer
@SpringBootApplication

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

Un ruolo importante gioca il file proprietries all'interno del percorso src/main/resources, in questo file specificheremo alcune logiche per l'avvio del Server e la sua confgurazione.
Come primo parametro imposteremo la porta di ascolto, dal momento che piu servizi possono essere in piedi nell'host in questione cercheremo di non interferire con altri servizi.
Come comportamento di Default quando il registro Eureka si avvia notificherà il fatto che non ci saranno nodi replica del registro a cui connettersi poiché infatti,in ambiti industriali vi saranno più istanze del registro, è necessario disabilitare questa opzione.
Un altro comportamento di default del registro Eureka è quello di registrare se stesso, per il nostro esempio non è necessaria questa modalità.

Per modificare queste dinamiche di Default e sepcificare i parametri di avvio del Server Eureka modificheremo il file application.properties nel seguente modo.

Nel file application.proprierties:

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false

logging.level.com.netflix.eureka=OFF
logging.level.com.netflix.discovery=OFF
```

Lato Client

Per quanto riguarda lato Client Eureka avremo un medesimo progetto Spring Boot di partenza.
Per poter interrogare il Servizio di regitrazione di Eureka è necessario spcificare la notazione @EnableDiscoveryClient che ci permetterà di utilizzare l'implementazione di DiscoveryClient.
È possibile specificare una diversa implementazione del per altri serivzi di registrazione come Hashicorp's Consul o Apache Zookeeper.

```
@EnableDiscoveryClient
@SpringBootApplication

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

Dal momento che il Client Eureka ha il copito di interagire con il Server avremo bisogno di impostare il Controller del container per poter dare la possibilità di interagire con l'utente tramite browser web.
Creeremo una classe a parte chiaamte ClientEurekaController.java.
In questa classe specificheremo che si tratta del controller attraverso l'inserimento della notazione @RestController.
All'interno di questa classe specificheremo il metodo con cui accedere da browser attraverso la notazione @RequestMapping.

```
@Autowired
private DiscoveryClient discoveryClient;

@RequestMapping("/service-instances/{applicationName}")
public List<ServiceInstance> serviceInstancesByApplicationName(
   @PathVariable String applicationName) {
         return this.discoveryClient.getInstances(applicationName);
               }
```

Attraverso la notazione @Autowired della classe DiscoveryClient permette di ridurre la quantità di configurazione necessaria alle iniezine dele dipendenze per il costruttore di classe DiscoveryClient.
Attraverso il parametro tra parentiesi graffe {applicationName} è possibile associare il nome dell'appicazione che ne fa uso con il nome specificato nel parametro spring.application.name che identifica il nome dell'applicazione.
In Spring Cloud questo parametro è usato spesso, questo proprietà è usata nel bootstrap del servizio e per convenzione risiede nel file bootstrap.proprerties nella cartella dove risiede il file application.properties.

```
spring.application.name=esmpionome
```

Il client Eureka definisce Un endpoint Spring MVC REST chiamato ServiceIstanceRestController che ritorna il numero di tutte le istanze di ServiceIstance registrate nel servizio di registrazione.
Per testare l'applicazione avviamo il Server di registrazione e successivamente il Client Eureka.
Il cliente impigherà del tempo per registrarsi al Server e successivamente tramite il browser è possibile interrogare il Server per avere come risposta l'istanza del servizio associato al client eureka .
Otterremo come risposta un file xml con i parametri relativi al client registrato in precedenza.

```
INFO 10952 --- [           main] c.e.E.EurekaClientApplication            : Started EurekaClientApplication in 13.55 seconds (JVM running for 15.49)
INFO 10952 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_SANPRONIO/LAPTOP-U0QUUUFE:sanpronio:8090 - registration status: 204
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : Disable delta property : false
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : Single vip registry refresh property : null
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : Force full registry fetch : false
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : Application is null : false
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : Registered Applications size is zero : true
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : Application version is -1: false
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : Getting all instance registry info from the eureka server
INFO 10952 --- [freshExecutor-0] com.netflix.discovery.DiscoveryClient    : The response status is 200
```

6.0 BILANCIAMENTO DEL CARICO CON RIBBON

Ribbon è un progetto integrabile con Spring sviluppato da Netflix OSS che fornisce un servizio di load balancing di tipo client-side.
Inoltre il progetto si integra automaticamente con il servizio di service discovery (Eureka) e con strumenti per la resilienza /Hystrix)
Ribbon provvede alle seguenti funzionalità:

- Regole per il bilanciamento del ccarico multiple e integrabili
- Integrazione con servizi di Discovery
- Tolleranza ai guasi
- Supporto a più protocolli (http, TCP, UDP) in un modello asincrono e con elevata reattività.
- Client integrati con il loadbalancer
- Configurazioni tramite Archaius
- Caching and batching

Moduli Principali

Ribbon è composta da 3 diversi moduli:

- ribbon-eureka:
  include implementazioni per il bilanciamento del carico basate su l Client Eureka, che è la libreria per iul servizio di registrazione e discovery.
- ribbon-httpclient:
  Include l'implementazione basata su JSR-311 di client Rest integrati con il bilanciatore del carico
  Client Rest costruito sopra ad Apache HttpClient integrato con il bilanciatore di carico
- ribbon-core:
  Include definizioni di interfacce per il bilanciamento del carico e per il client, implementazioni comuni ddel bilanciamento del carico e Integrazione del client con il bilanciatore

Attualmente il progetto Ribbon è in fase di sviluppo pertanto alcune delle tecnologie e metodologie descritte potrebbe risultare deprecato in un momento successivo.

6.1 Client Ribbon

L'applicazione Client sarà il servizio che peretterà agli utenti di vedere i risultati, effettuerà la chiamata ai Server per avere una risposta e successvamente inoltrarla all'utente finale che legge per esempio tramite browser all'endpoint /hi.
Partendo come base anche qui da un progetto Spring Boot, implementeremo la classe princicpale RibbonClientApplication.

Anche questa classe verrà annotata da @RestController, inoltre per effettuare chiamate remote http utilizzeremo la classe RestTemplate.
RestTemplate fa una chiaata http di tipo GET al server e riceve una stringa come risposta

```
    @Bean
    RestTemplate restTemplate(){
        return new RestTemplate();
      }
    @Autowired
    RestTemplate restTemplate;
```

Ora è necessario abilitare Ribbon lato Client e per farlo modificheremo il file application.yml inserendo le seguenti proprietà:

```
spring:
  application:
    name: user
server:
  port: 8888
say-hello:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:8090,localhost:9092,localhost:9999
       ServerListRefreshInterval: 15000
```

Come prima cosa settiamo il nome dell'applicazione lato utente e impostiamo il servizio in ascolto sulla porta 8888.
Sprig Cloud Netflix crea un ApplicationContext per ogni nome elencato in questo file che rappresenta ogni Client Ribbon nell'applicazione.
È usato per dare a un Client un set di beans per le istanze dei componenti Ribbon.
Nel nostro caso sopra il client è chiamato say-hello, e le proprietà che abbiamo settato sono eureka.enabled(il quale lo settiamo a false), listofServers, e ServerListRefreshInterval.
Normalmente Il bilanciatore di carico Ribbon prende la lista dei Server da un tipico Servizio di registro Eureka.

ServerListRefreshInterval è il parametro dell'intervallo del tempo che passa per controllare nuovamente la lista, espresso in millisecondi.

```
@SpringBootApplication
@RestController
@RibbonClient(name = "say-hello", configuration = SayHelloConfiguration.class)
public class RibbonClientApplication {

    @LoadBalanced
      @Bean
      RestTemplate restTemplate(){
        return new RestTemplate();
      }

      @Autowired
      RestTemplate restTemplate;

      @RequestMapping("/hi")
            public String hi(@RequestParam(value="name", defaultValue="Artaban")
    String name) {
```

```
            String greeting = this.restTemplate.getForObject("http://say-
    hello/greeting", String.class);
            return String.format("%s, %s!", greeting, name);
          }

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

Modifichiamo il file RibbonClientApplication.java inserendo la notazione @LoadBalanced sopra al metodo “RestTemplate()” , questo dice a Spring Cloud che noi vogliamo usufruire del supporto del Bilanciatore del carico.
Questo supporto viene dato dalla classe RibbonLoadBalancerClient che ci permetterà di interagire con il servizio.
A sua volta, questo permette di usare "identificatori logici" per gli URL che passi al RestTemplate.
Questi identificatori logici sono in genere il nome di un servizio.
per esempio:

```
restTemplate.getForObject("http://example/user/{id}", String.class, 1);
```

dove “example” è il nome logico.

I componenti di un Bilanciatore di carico sono:

- Regole:una componente logica per determinare quale server prendere da una lista
- Ping:componente eseguita in background per assicurarsi il corretto stato di vita dei server
- Lista dei Server:lista dei server che puo essere statica o dinamica, un processo eseguito in background aggiornerà e filtrerà la lista a un certo intervallo.

Queste componenti possono essere sia impostate programmaticamente o essere parte delle proprietà della configurazione del Client e create tramite Reflection.
I nomi delle relative proprietà devono essere precedute da `<clientName>.<nameSpace>` `all’interno del file .propreties` nel classpath del progetto e sono:

- `NFLoadBalancerClassName`
  dovrebbe implementare `IloadBalancer`
- `NFLoadBalancerRuleClassName`
  Dovrebbe implementare `IRule`
- `NFLoadBalancerPingClassName`
  Dovrebbe implementare `IPing`
- `NIWSServerListClassName`
  Dovrebbe implementare `ServerList`
- `NIWSServerListFilterClassName`
  Dovrebbe implementare ServerListFilter

Esempio:
Per impostare una regola per un servizio chiamato users si inserire nel file .yml:

```
users:
  ribbon:
       NFLoadBalancerRuleClassName:com.netflix.loadbalancer.WeightedResponseTimeRule
```

Le classi definite da queste proprietà hanno la precedenza sui bean definiti usando @RibbonClient(config.class) e quelli predefiniti provvisti da Spring Cloud Netflix.

Le regole del LoadBalancer messo a disposizione da Ribbon sono:

- **RoundRobinRule**:

Queste regole scelgono i server secondo RoundRobin. Sono spesso usate come regole di default.

- **AvailabilityFilteringRule**:
  Questa regola salterà i server che sono considerati "circuit tripped" o con alto numero di connessioni simultanee.
  Per impostazione predefinita, un'istanza viene dette “CircuitTrpped” se il Client Rest non riesce a stabilire una connessione per le ultime tre volte. Una volta che un'istanza è “CircuitTripped”, rimarrà in questo stato per 30 secondi prima che il circuito venga ritenuto nuovamente chiuso. Tuttavia, se la connessione continua a fallire, scatterà nuovamente il circuito e il tempo di attesa per farlo diventare chiuso aumenterà esponenzialmente al numero di guasti consecutivi. Esempio:

  `niws.loadbalancer.<clientName>.connectionFailureCountThreshold`:
  Soglia di fallimento delle successive connessioni per mettere il server in stato “Circuit Tripped”,di default sono 3

  `niws.loadbalancer.<clientName>.circuitTripMaxTimeoutSeconds`:
  Periodo massimo che un istanza puo rimanere in stato “inutilizzabile” indipendentemente dall’incremento esponenziale prodotto dai fallimenti, di default sono 30 secondi

  `<clientName>.<clientConfigNameSpace>.ActiveConnectionsLimit`:
  Soglia per il numero di connessioni simultanee prima di saltare il server, di default si usa Integer.MAX_INT.

- **WeightedResponseTimeRule**:
  Secondo questa regola, a ciascun server viene assegnato un peso in base al tempo di risposta medio. Più lungo è il tempo di risposta, minore sarà il peso che otterrà. La regola seleziona un server in base al peso posseduto.
- **Random:**

- **Cone Aware:**
  Il LB calcola7esamina lo stato delel zone in real time.
  Se la media delle richieste attive raggiunge una certa soglia la zona verrà eliminata

La classe è annotata da @RibbonClient il quale ci da la possibilità di configurare il Client Ribbon.

```
@RibbonClient(name = "say-hello", configuration = SayHelloConfiguration.class)
```

- **name** – necessario settarlo con lo stesso nome del servizio che si sta chiamando con ribbon, necessita di ulteriore configurazione per come Ribbon interagisce con il servizio.
- **configuration** – ci da la possibilità di indicare una classe java usata per la configurazione difinita come @Beans. Bisogna esser sicuri che questa classe non sia annotata da @ComponentScan atrimenti sovrascriverà i valori predefiniti per tutti i client Ribbon

La notazione @RibbonClient normalmente non viene richiesta, ci sono due casi in cui la notazione è richiesta:

1. necessita di customizzare le impostazioni di Ribbon per un particolare Client ribbon,
2. se non si usa nessun servizio di discovery

Spring Cloud usa la notazione @RibbonClient per configurare i tipi usati da ribbon come le liste dei server.
Se si ha Eureka nel classpath di default usa la lista dei server da Eureka.

Ribbon provvede di andare a reperire la lista deiserver in modo dinamico tramite Eureka come comportamento di default.
Nel nostro caso non verrà utilizzato Eureka ma una lista statica di server predefiniti.

Per disabilitare Eureka e aggiungere una lista statica definita a priori basta modificare il file yml aggiungendo :

```
eureka.enable=false
listOfServers: localhost:8090,localhost:9092,localhost:9999
```

Nel caso si voglia applicare una configurazione a tutti i Client Ribbon del nostro server è possibile utilizzare la notazione `@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)`, dove DefaultRibbonConfig.class è la nostra classe Java.

Ora quindi avremo bisogno di definire la nuova classe chiamata SayHelloConfiguration.java nel percorso lato Client "src/main/java/" usata per definire i nostri @Bean.

```
public class SayHelloConfiguration {

  @Autowired
  IClientConfig ribbonClientConfig;

  @Bean
  public IPing ribbonPing(IClientConfig config) {
    return new PingUrl();
  }

  @Bean
  public IRule ribbonRule(IClientConfig config) {
    return new AvailabilityFilteringRule();
  }

   }
```

Attraverso questa configurazione è possibile ridefinire una serie di bean chiamandoli con lo stesso nome.

Spring Cloud Netflix provvede a una serie di Bean di default per Ribbon `<Tipobean` nomeBeam: `NomeClasse>`.

- `IClientConfig` ribbonClientConfig: `DefaultClientConfigImpl`
  Definisce la configurazione del client utilizzata dalle varie API per inizializzare i client o i bilanciatori di carico
- `IRule` ribbonRule: `ZoneAvoidanceRule`
  Interfaccia che definisce una "Regola" per un LoadBalancer. Una regola può essere pensata come una strategia per il loadbalacing. Le più note strategie di bilanciamento del carico includono Round Robin, Response Time Based ecc.
- `IPing` ribbonPing: `DummyPing`
  Interface that defines how we "ping" a server to check if its alive
  Iterfaccia che deifnisce come noi inviamo "ping" a server per controllare lo stato di salute
- `ServerList<Server>` ribbonServerList: `ConfigurationBasedServerList`
  Interfaccia che definisce i metodi utilizzati per ottenere l'elenco dei server
- `ServerListFilter<Server>` ribbonServerListFilter: `ZonePreferenceServerListFilter`

Questa interfaccia consente di filtrare l'elenco dei server che possono essere ottenute da una lista statica o Dinamica nel caso di Eureka, con caratterisctiche desiderabili.

- `ILoadBalancer` ribbonLoadBalancer: `ZoneAwareLoadBalancer`
  Interfaccia che definisce le operazioni per un software di bilanciatore di carico.
  Un loadbalancer tipicamente richiede un insieme di server per cui bisogna effettuare il bilanciamento del carico (**void** `addServers(List<Server> newServers)`,un metodo per contrassegnare un particolare server fuori dalla rotazionione definita dalla strategia di bilanciamento `(void markServerDown(Server server))` e una chiamata che sceglierà un server da un elenco esistente `( Server chooseServer(Object key))`.
- `ServerListUpdater` ribbonServerListUpdater: `PollingServerListUpdater`
  Interfaccia che da la possibiltà di usare differenti strategie per fare aggiornamenti dinamici della lista dei server.

Per testare il sistema creato basta eseguire i 3 server rispettivamente sulle porte 8090, 8692 e 9999, eseguire poi Il client Ribbon sulla porta 8888 e testare tramite il browser all'indirizzo /hi che si riceva correttamente la risposta del server Ribbon.
Da notare che il ping di Ribbon avviene ogni 15 secondi.
In questo modo la richiesta effettuata si distribuisce sui vari server attraverso la strategia round robin.
Se si prova a terminare uno dei 3 server Ribbon si nota che saranno disponibili gli altri due che accetteranno la nostra richiesta e risponderanno correttamente.

6.2 Funzionamento con Eureka

Quando Eureka viene utilizzato insieme a Ribbon, `ribbonServerList` viene sostituita con un'estensione di `DiscoveryEnabledNIWSServerList` che popola l'elenco di server dai Server Eureka.
Viene sostituita anche l'interfaccia IPing con NIWSDiscoveryPing il quale delega a Eureka il compito di determinare se un server è attivo.
Di default la lista dei server viene costruita attraverso informazioni sulla "zona", previste nei metadati dell'istanza, (eureka.instance.metadataMap.zone), e se questi vengano a mancare puo usare il nome del dominio da un Server come un proxy di zona (se il flag approximateZoneFromHostname è settato).
Una volta che le informazioni sulla zona sono disponibili, possono essere utilizzate in un ServerListFilter.
Per impostazione predefinita verrà utilizzato per individuare un server nella stessa zona del client poiché il valore predefinito è ZonePreferenceServerListFilter. La zona del client è determinata allo stesso modo delle istanze remote per impostazione predefinita, ovvero tramite eureka.instance.metadataMap.zone.

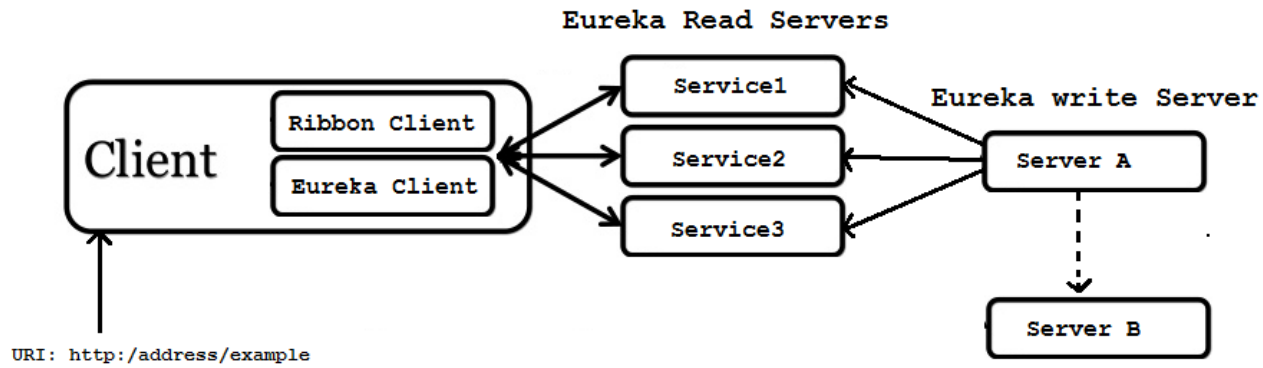


Fig.15

6.3 ServerListFilter

Componente usata da DynamicServerListLoadBalancer per filtrare i serrver da ritornare dalla lista.
Ci sono 2 implementazioni di ServerListFilter in Ribbon:

- ZoneAffinityServerListFilter
  Filtra i server che non si trovano nella stessa zona del client, a meno che non ci siano server disponibili nell'area client.
  Questo filtro può essere abilitato specificando le seguenti proprietà (assumendo che il nome del client sia “myClient” e che il nemaspace delle proprietà del client sia “ribbon”)
  Es.
  ```
  myclient.ribbon.EnableZoneAffinity=true
  ```

- ServerListSubsetFilter
  Questo filtro si assicura che il client veda solo un sottoinsieme fisso di server generali restituiti dall'implementazione ServerList. Può anche sostituire periodicamente i server nel sottoinsieme di scarsa disponibilità con nuovi server. Per abilitare questo filtro, specificare le seguenti proprietà:

  ```
  myClient.ribbon.NIWSServerListClassName=com.netflix.niws.loadbalancer.D
  iscoveryEnabledNIWSServerList
  ```

  Il server deve registrarsi con Eureka server con VipAddress “myservice”
  ```
  myClient.ribbon.DeploymentContextBasedVipAddresses=myservice
  myClient.ribbon.NIWSServerListFilterClassName=com.netflix.loadbalancer.
  ServerListSubsetFilter
  ```

  Mostra 5 servizi clienti di default sono 20.
  ```
  myClient.ribbon.ServerListSubsetFilter.size=5
  ```

6.4 Utilizzo di Namespace

Il modo piu facile per configurare il client e il bilanciatore del carico è attraverso il caricamento delle proprietà in Archaius che è conforme al seguente formato:

```
<clientName>.<nameSpace>.<propertyName>=<value>
```

É possibile definire proprietò in un file sel classpath o nelle proprieta di sistema,
Se formato, attraverso “ConfigurationManager.loadPropertiesFromResources()” è possibile caricare il file.
Di default “ribbon “ deve essere il namespace.
Se non vi sono proprietà specificate per un nome Client, “ClientFactory” creerà il client e il bilanciatore del carico con valori di default per tutte le propeietà necessarie.
I valori di Default sono specificati in “DefaultClientConfigImpl”.
Se una proprietà manca il nome del cliente viene interpretato da Ribbon come ina proprietà da estendere a tutti i Client.
Per esempio:

```
ribbon.ReadTimeout=1000
```

È anche possibile impostare le proprietà a livello di codice costruendo l'istanza di DefaultClientConfigImpl.

- Chiamare “DefaultClientConfigImpl.getClientConfigWithDefaultValues(String clientName)” per caricare i vaoli predefiniti e ogni altra proprietà definita nella cnfigurazione in Archaius.

- Settare le proprietà desiderate richiamando DefaultClientConfigImpl.setProperty()

Namespace ClientFactory

Se si vuole utilizzare un namespace per le properietà differente diverso da “ribbon” e utilizzare le API ClientFactory per costruire il Client o il bilanciatore del carico ci sono diversi modi:

1-Estendere la classe “DefaultClientConfigImpl” e sovrascrivere il metodo “getNameSpace()”

```
public class MyClientConfig extends DefaultClientConfigImpl {
             // ...
             public String getNameSpace() {
                 return "foo";
             }
               }
```

Assumendo che le proprietà sono definite come myclient.foo.* è possibile utilizzare le api ClientFactory per creare il Client:

```
MyClient client = (MyClient) ClientFactory.createNamedClient("myclient",
MyClientConfig.class);
```

2-Usare DefaultClientConfigImpl ma cambiare il namespace de default.

```
DefaultClientConfigImpl clientConfig = new DefaultClientConfigImpl("foo");
  clientConfig.loadProperites("myclient");
  MyClient client = (MyClient)
ClientFactory.registerClientFromProperties("myclient", clientConfig);
```

6.5 Accedere alle risorse con RibbonRequest

È possibile accedere alle risorse di rete definendo le interfacce con le annotazioni di Ribbon, e ottenere un'istanza dell'interfaccia da Ribbon e richiamare i metodi come un normale oggetto Java.

```
public interface MovieService {
    @Http(  method = HttpMethod.GET,
            uri = "http://localhost:8080/users/{userId}/recommendations")
    RibbonRequest<ByteBuf> recommendationsByUserId(@Var("userId") String
userId);
}
```

La definizione dell'interfaccia dice a Ribbon di fare una richiesta GET all'URI specificato e sostituire {userId} con il valore effettivo di userId passato nella chiamata al metodo.
L’applicazione puo creare un istanza di MovieService e invocare i supoi metodi.

```
movieService = Ribbon.from(MovieService.class);
    Observable<ByteBuf> result =
movieService.recommendationsByUserId(TEST_USER).toObservable();
```

6.6 Utilizzo del RequestTemplate

Utilizzando le API di Ribbon è possibile accedere a risorse http (tipicamente un REST endpoint in un server come http://examplehost:/foo) attraverso RequestTemplate

Creazione di HttpResourceGroup

STEP1:

```
HttpResourceGroup httpResourceGroup =
Ribbon.createHttpResourceGroup("movieServiceClient",
            ClientOptions.create()
                    .withMaxAutoRetriesNextServer(3)
                    .withConfigurationBasedServerList("localhost:" +
RxMovieServer.DEFAULT_PORT));
```

HttpResourceGroup viene utilizzato per gestire attributi comuni (come ClientOptions) che saranno poi condivisi da HttpRequestTemplate.

Creare un HttpRequestTemplate da HTTpResourceGroup

STEP 2

```
HttpRequestTemplate<ByteBuf> recommendationsByUserIdTemplate =
httpResourceGroup.newTemplateBuilder("recommendationsByUserId",
ByteBuf.class)
            .withMethod("GET")
            .withUriTemplate("/users/{userId}/recommendations")
            .withHeader("X-Auth-Token", "abc")
            .withFallbackProvider(new RecommendationServiceFallbackHandler())
            .withResponseValidator(new
RecommendationServiceResponseValidator())
            .build();
```

In questo passaggio, è possibile specificare il metodo HTTP, il modello URI, i provider di cache, il fallback di Hystrix e il validatore di risposta per una particolare risorsa HTTP. Il validatore di risposte è responsabile per il controllo della risposta HTTP e fornisce l'indicazione a Hystrix se deve essere offerto un fallback.
Se si usa il nome della variabile come "{id}" invece del suo valore reale, queste verranno successivamente sostituite con valori reali quando viene creata la richiesta effettiva in realtime.

Creare un RequestBuilder

STEP 3

```
RibbonRequest<ByteBuf> request =
recommendationsByUserIdTemplate.requestBuilder()
                      .withRequestProperty("userId", TEST_USER)
                      .build();
```

Invocare RibbonRequest API

STEP 4

```
Observable<ByteBuf> result = request.observe();
```

L'interfaccia RibbonRequest

Interfaccia che provvede a utilizzare metodi bloccanti e non bloccanti per recuperare il contenuto di una richiesta.

```
T execute()
```

API bloccante che ritorna un singolo elemento (o ultimo elemento se c'è una sequesza di oggetti da una singola esecuzione)

```
Observable observe()
```

API non bloccante che restituisce un oggetto Observable mentre l'esecuzione è avviata in modo asincrono

```
Observable toObservable()
```

API non bloccante che restituisce un Observable. L'esecuzione non viene avviata fino a quando l'Observable restituito non viene sottoscritto.

```
RequestWithMetaData withMetadata()
```

Crea un RequestWithMetaData in cui è possiblile chiamare le API bloccanti e non bloccanti per ottenere un oggetto RibbonResponse, che a sua volta contiene altri oggetti e alcuni metadati derivanti dall'esecuzione di Hystrix.

Pattern Observed

Il pattern **Observer**, in particolare, rappresenta una soluzione nei casi in cui è necessaria una certa relazione (**da uno a molti**) tra un oggetto caratterizzato da diversi stati interni, ed una serie di altri oggetti che reagiscono in conseguenza al cambimento di questi stati.
Quindi abbiamo un'oggetto che viene "osservato" (detto **observable**) e tanti oggetti che "osservano" i cambiamenti di quest'ultimo (detti **observers**).
Uno dei tanti vantaggi di questo schema è quello di limitare la dipendenza fra due tipi di oggetti, nel senso che ambedue le parti possono modificare la loro struttura senza avere un'impatto sull'altro, lavorando in maniera totalmente disaccoppiata.

6.7 Panoramica sui moduli di Ribbon

Descrizione generale sulle funzionalità delle singole classi e interfacce disponibili per quel particolare modulo.

Modulo Ribbon Core

Loadbalancer

SOMMARIO CLASSI

- AbstractLoadBalancer
  Classe che contiene caratteristiche richieste per la maggior parte delle implementazioni di loadbalancing.
  L'anatomia di un tipico LoadBalancer è costituita da
  1. Un elenco di server (nodi) che sono potenzialmente basati su un criterio specifico.
  2. Una classe che definisce e implementa una strategia di LoadBalacing tramite IRule
  3. Una classe che definisce e implementa un meccanismo per determinare l'idoneità/disponibilità dei nodi/server nell'elenco.
- AbstractLoadBalancerPing

Classe che provvede l'implementazione di base per determinare lo stato di "vita" di un Server(nodo)

- AbstractLoadBalancerRule
  Classe che fornisce un'implementazione predefinita con metodi get e set del LoadBalancer
- AbstractServerList<T extends Server>
  Classe che definisce come una lista di Server è ottenuta, aggiornata e filtraata per essere utilizzata da NIWS.
- AbstractServerListFilter<T extends Server>
  Classe che è responsabile di filtrare l'elenco dei server da quelli attualmente disponibili dal Load Balancer
- AvailabilityFilteringRule
  Una regola del bilanciamento del carico che filtra I server che hanno connessioni attive che superano un limite già confgurato(di default è Integer.MAX_VALUE).
- BaseLoadBalancer
  Un'implementazione di base del servizio di bilanciamento del carico in cui è possibile impostare un elenco arbitrario di server come pool di server. È possibile impostare un ping per determinare lo stato di un server. Internamente, questa classe mantiene una lista di server chiamata "all" e una lista di server chiamata "up" e li usa a seconda di cosa richiede il choamante.
- ClientConfigEnabledRoundRobinRule
  Questa classe contiene essenzialmente la classe RoundRobinRule definita nel pacchetto loadbalancer
- ConfigurationBasedServerList
  Classe di utilità in grado di caricare l'elenco di server da una configurazione (ovvero le proprietà disponibili tramite Archaius)
- DummyPing
  Implementazione semplice e di Dfault che marca lo stato di un Server
- DynamicServerListLoadBalancer<T extends Server>
  Un LoadBalancer che ha le capacità per ottenere l'elenco dei server candidati utilizzando una sorgente dinamica. Ad esempio, l'elenco dei server può essere modificato in fase di runtime.
- Interrupt Task:
  implementazione dell'interrupt
- LoadBalancerStats
  Classe che funge da archivio di caratteristiche operative e statistiche di ogni Nodo / Server in LaodBalancer. Queste informazioni possono essere utilizzate per osservare e comprendere il comportamento di runtime del loadbalancer o, ancora più importante, per la base che determina la strategia loadbalacing
- NoOpLoadBalancer:
- NoOpPing
- PingConstant
  Un utility dell'implementazione di Ping che puo ritornare "alive" o "dead"
- RandomRule
  Una strategia di bilanciamento del carico che distribuisce in modo casuale il traffico tra i server esistenti.
- ResponseTimeWeightedRule
  Regola che utilizza i tempi di risposta sotto forma di media o percentuale per assegnare "pesi" dinamici per Server
- RetryRule
  Dato che IRule può essere collegato in cascata, questa classe RetryRule consente di aggiungere una logica di ripetizione a una regola esistente.
- RoundRobinRule
  La più nota e basilare strategia di bilanciamento del carico, vale a dire Regola del Robin.

- Server
  Classe che rappresenta un tipico server (o un nodo indirizzabile), esempio: host: identificatore di porta
- ServerComparator
  Classe per aiutare a stabilire l'uguaglianza per le operazioni di hash / chiave
- ServerStats
  Cattura varie statistiche del Server (nodo) nel LoadBalancer
- WeightedResponseTimeRule
  Questa classe contiene essenzialmente la classe ResponseTimeWeightedRule definita nel pacchetto loadbalancer
- ZoneAvoidanceRule:
  Una regola che utilizza un oggetto "CompositePredicate" per filtrare i server in base alla zona e alla disponibilità.
- ZoneAwareLoadBalancer<T extends Server>
  Load Balancer che può evitare un intera zona quando si sceglie il server.
  La metrica utilizzata per misurare la condizione della zona è Average Active Requests.Sono le richieste totali in sospeso in una zona divisa per il numero di istanze disponibili.
  Il LoadBalancer calcolerà ed esaminerà le statistiche di zona di tutte le zone disponibili. Se le Richieste attive medie per qualsiasi zona hanno raggiunto una soglia configurata, questa zona verrà eliminata dall'elenco dei server attivi. Nel caso in cui più di una zona abbia raggiunto la soglia, la zona con le richieste più attive per server verrà eliminata. Una volta che la zona peggiore viene eliminata, verrà scelta una zona tra le altre con la probabilità proporzionale al numero di istanze. Un server verrà restituito dalla zona, scelta con una determinata regola. (Una regola è una strategia di bilanciamento del carico, ad esempio AvailabilityFilteringRule).
- ZoneSnapshot
  cattura le metriche per zona
- ZoneStats<T extends Server>
  Classe che salva le statistiche per Zona (dove la zona è tipicamente Amazon Availability Zone)

Client

Classi:

- AbstractLoadBalancerAwareClient<S extends ClientRequest,T extends IResponse>
  Classe astratta che fornisce l'integrazione del client con i bilanciatori del carico.
- ClientFactory
  Classe che crea istanze del client, del bilanciatore del carico e configurazione dalle proprietà. Mantiene anche i mapping dei nomi dei clienti alle istanze create.
- ClientRequest
  Un oggetto che rappresenta una richiesta client comune adatta a tutti i protocolli di comunicazione.
- PrimeConnections
- SimpleVipAddressResolver
  Un Vip Address nella terminologia Ribbon è un nome logico usato per specificare un Server

Interfacce:

- IClient<S extends ClientRequest,T extends IResponse>
  Interfaccia per l'esecuzione da parte di un Client di una singola richiesta
- IclientConfigAware
  Esistono più classi (e componenti) che richiedono l'accesso alla configurazione.
- IPrimeConnection

Interfaccia che definisce l'operazione per l'avvio di una connessione.

- IResponse
  Interfaccia di risposta per il framework clientw
- PrimeConnections.PrimeConnectionListener
- VipAddressResolver
  VipAddress per una specifica server farm

Client.config

Interfacce:

- IClientConfig
  Definisce la configurazione del client utilizzata dalle varie API per inizializzare i client o caricare i bilanciatori.
- IClientConfigKey
  Definisce la chiave utilizzata in IclientConfig

Classi:

- DefaultClientConfigKey
  Configurazione del client predefinita che carica le proprietà dal ConfigurationManager di Archaius.

6.8 Modulo Ribbon Eureka

Loadbalancer

Classi:

- DefaultNIWSServerListFilter
  Il filtro NIWS predefinito: si occupa di filtrare i server in base all'affinità di Zone e ad altre proprietà correlate
- DiscoveryEnabledNIWSServerList
  Classe per contenere un elenco di server che il RestClient NIWS può utilizzare
- DiscoveryEnabledServer
  Server che sono stati ottenuti tramite Discovery e quindi contengono metadati sotto forma di InstanceInfo
- NIWSDiscoveryPing
  "Ping" Discovery Client, non è un vero "ping". Viene dedotto che il server sia attivo se Discovery Client lo dice.

6.9 Modulo Ribbon HttpClient

Http4

Classi:

- ConnectionPoolCleaner
  Classe responsabile della pulizia delle connessioni in base a una politica Ad es. rimuovere tutte le connessioni dal pool che sono rimaste inattive per più di x msecs
- NFHttpClient
  Estensione Netflix di Apache 4.0 HttpClient

- NFHttpClientConstants
- NFHttpClientFactory
  classe usata per ottenere un'istanza di NFHttpClient
- NFHttpMethodRetryHandler
  Specifica calsse di estensione per DefaultHttpMethodRetryHandler

Loadbalancer

Classi:

- PingUrl
  Implementazione del ping se si desidera eseguire un "controllo dello stato di salute" su un server. Sara un Ping reale a differenza di alcuni servizi / clienti che scelgono PingDiscovery - che è veloce ma non è un vero ping perché si basa su una risposta non ottenuta direttamente ma delegando il nodo in questione

Niws.cert

Classi:

- AbstractSslContextFactory
  Classe astratta per rappresentare ciò che logicamente associamo al contesto ssl sul lato client, ovvero il keystore e il truststore.

Niws.client

Classi:

- URLSslContextFactory
  Factory di socket che utilizza il codice NIWS se viene specificato un keystore o un trust store non di default.

Niws.client.http

Classi:

- HttpClientRequest:
  classe per gestire le richioeste http del CLient
- HttpClientRequest.Builder
- HttpClientResponse
  Risposta del client NIWS (questa versione include solo la risposta del client Jersey)
- RestClient
  modulo che è un wrapper per il client Jersey.
  Per impostazione predefinita, utilizza HttpClient per le comunicazioni HTTP sottostanti.
  L'applicazione può impostare il proprio client Jersey con questa classe, ma così facendo annullerà tutte le configurazioni client impostate in IClientConfig.

7.0 DESCRIZIONE DEL PROGETTO

CONSIDERAZIONE :
questo file documenta in linea generatle le configurazioni usate nel progetto, seppure funzionanti sono mancanti di alcuni dettagli non ancora inseriti come il tempo di ricezione degli heartbeats, abilitazione o no del self registration mode, disattivazione della sicurezza per le richieste etc..
Questo è per il motivo che solo dopo aver testato il sistema si avrà piu una vision chiara sui dettagli da finire di implementare nel file di configurazione.

Per la relizzazione del progetto è stato utilizzato l'IDE Eclipse EE e come plugin GRADLE STD e MAVEN il quale permette di importare rispettivamente progetti che utilizzano un file build.gradle o pom.xml.
Per la relizzazione della pagina web relativa all'utente è stato utilizzato l'IDE NetBeans.
Azione necessaria al fine del corretto funzionamento del sistema è la modifica del file etc/hosts il quale dovrà essere preventivamente modificato per poter simulare il comportamento dei Server Eureka.
Infatti Eureka utilizza i nomi di host per poter dirigere le richieste.
Per la simulazione in locale il file modificato risultante sarà con l'aggiunta delle seguenti stringhe:
127.0.0.1 peer1
127.0.0.1 peer2
127.0.0.1 peer3
localhost peer1
localhost peer2
localhost peer3

7.1 Lato Server (Eureka peer1,peer2, peer3)

Applicazione Spring su cui viene aggiunta la notazione @EnableEurekaServer la quale ci dice che il server viene marcato come registro.
Dal momento che questo registro contiene tutte le informazioni sui servizi reperibili è necessario mantenere un elevata disponbibiltà.
Oltre al Server principale sono stati implementati altri 2 server che fungono da Buckup delle informazioni qualora non fosse più raggiungibile centrale.
I Server agiscono come una rete di peer to peer, periodicamente si inviano segnali per mantere aggiornato il servizio.
Nel caso venga a meno il Server1 è quindi possibile mantenere le informazioni sui servizi senza perderne il contenuto.

La configurazione dei Server Eureka sono nel file application.yml

```
---
spring:
  profiles: peer1
server:
  port: 8761
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
```

```
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/
---
spring:
  profiles: peer2
server:
  port: 8762
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/
---
spring:
  profiles: peer3
server:
  port: 8763
eureka:
  instance:
    hostname: peer3
  client:
    serviceUrl:
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/
```

7.2 Lato Client (User Service)

Da notare che I servizi registrati nel registro Ereka fungono sia da Client che da server , inizialmente in fase di registrazione si conporta da Client mandando le richieste al server centrale, successivamente si comporterà da server rispetto a Servizio degli utenti che dovranno inviare loro le richieste per poter consultare i servizi.
Questo servizio online di Video mette a disposizione una collezzione di articoli che è possibile consultare e successivamente acquistare.
Il servizio VideoService invia richiesta di registrazione al Server Eureka 1 il quale lo memorizza nel registrao e si preoccuparà di mantenere aggiornato lo stato dei servizi in base al proprio pool.
La classe principale viene annotata con `@EnableDiscoveryClient` e verrà istanziato il load balancer attraverso:

```
    @Autowired
    private LoadBalancerClient client;

    @Autowired
    private RestTemplate restTemplate;

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

File ClientApplication.java:

```
      @Bean
      public InternalResourceViewResolver viewResolver() {
```

```
            InternalResourceViewResolver viewResolver = new
InternalResourceViewResolver();
            viewResolver.setViewClass(JstlView.class);
            viewResolver.setPrefix("/WEB-INF/jsp/");
            viewResolver.setSuffix(".jsp");
            return viewResolver;
      }


      public static void main(String[] args) {
            SpringApplication application = new
SpringApplication(SpringBootJspExampleApplication.class);
            application.setAdditionalProfiles("client");
            application.run(args);
      }
```

Con il primo metodo “viewResolver” è possibile tramite codice impostare il percorso relativo alla pagina jsp per l’utente.

File application.yml:

```
spring:
  profiles: client
  application:
    name: client
server:
  port: 9001
management.security.enabled: false
eureka:
  client:
    service-url:
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/
  instance:
    lease-expiration-duration-in-seconds: 2
    lease-renewal-interval-in-seconds: 1
```

I metodi della classe WelcomeController.java:

```
@RequestMapping(value = "/Homes", method = RequestMethod.GET)
          public String viewRegister3(Map<String, Object> model,ModelMap mod) {
              User userForm = new User();
              model.put("userForm2", userForm);



              String s="server";
             List<String> myList = new ArrayList();
             List<String> myList2 = clients.getServices();
             for (String ob : myList2) {
                         if (ob.substring(ob.length()-6,ob.length()).equals(s)){
                               myList.add(""+ob+"");
                         }
                   }
              ArrayList<String> sa=new ArrayList<String>();
              sa.add("some string1");
              sa.add("some string2");
              sa.add("some string3");
               mod.addAttribute("modules",myList);
```

```
            return "home";
        }

    @RequestMapping(value="/fetch/{id}",method = RequestMethod.GET)
    @ResponseBody
    public String getInfo(@PathVariable("id") String id) {
        System.out.println("id:"+ id);

        String name = restTemplate.getForObject("http://"+id+"/name", String.class);
     logger.info(name.toString());



        return name;
    }
```

I metodi principali sono:
“viewRegister3” : ritorna un file .jsp dove è possibile consultare i servizi disponibili in quel momento e registrati in Eureka Server Registry.
“getInfo”:metodo che permette di effettuare le chaimate ai servizi conosciuti dal registro, attraverso questo metodo viene utilizzato il load balance per effettuare le richieste.

La pagina web esposta dal metodo viewRegister3 è definita come segue nel percorso src/main/webapp/WEB-INF/jsp/:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
List of avaible services: <br>${modules}
<br>
<br>
<br>
<br>
For access to services type "/fetch/{service-name}" from Url

</body>
</html>
```

7.3 Lato Client (Video Service)

Servizio di Video che mette a disposizione una pagina web un controller:

```
@RequestMapping("/name")
    public ArrayList<String> text2() {

      ArrayList<String> lista=new ArrayList<String>();
      lista.add("[S1]Articolo 1");
      lista.add("[S1]Articolo 2");
      lista.add("[S1]Articolo 3");
```

```
        return lista;
    }
```

Come esempio si è voluto mostrare il contenuto di una lista statica creata appositamente per verificare il funzionamento del controller con ribbon.
Qualora si volesse si puo utilizzare un database come postgres,Mongodb e altri.
File Application.yml:

```
spring:
  profiles: server-1
  application:
    name: Video-server
server:
  port: 8001
management.security.enabled: false
eureka:
  client:
    service-url:
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/

  instance:
    lease-expiration-duration-in-seconds: 2
    lease-renewal-interval-in-seconds: 1
---
spring:
  profiles: server-2
  application:
    name: Video-server
server:
  port: 8002
management.security.enabled: false
eureka:
  client:
    service-url:
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/

  instance:
    lease-expiration-duration-in-seconds: 2
    lease-renewal-interval-in-seconds: 1
```

Il secondo profilo specificato nel file yml è relativo alla replica del servizio in questione, qualora ci fosse la necessità sarebbe possibile mantenere una copia del server per dare una elevata disponibilità in caso di guasti o imprevisti.
Nel file di configurazione application.yml viene specificato il nome dell'applicazione, questo nome è utilizzato da ribbon per gestire le richieste da effettuare.

7.4 Lato Client (Book Service)

Servizio che mette a disposizione una vasta gamma di libri consultabili e acquistabili.
Il metodo principale è "lis" Il quale espone una lista di prodotti consultabili dall'utente:

```
@RequestMapping("/name")
    public lis<String> text2() {
```

```
        ArrayList<String> lista=new ArrayList<String>();
        lista.add("[S1]Articolo 1");
        lista.add("[S1]Articolo 2");
        lista.add("[S1]Articolo 3");

          return lista;
    }
```

File Application.yml:

```
spring:
  profiles: Bserver-1
  application:
    name: Book-server
server:
  port: 8021
management.security.enabled: false
eureka:
  client:
    service-url:
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/
  instance:
    lease-expiration-duration-in-seconds: 2
    lease-renewal-interval-in-seconds: 1
---
spring:
  profiles: Bserver-2
  application:
    name: Book-server
server:
  port: 8032
management.security.enabled: false
eureka:
  client:
    service-url:
      defaultZone:
http://peer1:8761/eureka/,http://peer2:8762/eureka/,http://peer3:8763/eureka/
  instance:
    lease-expiration-duration-in-seconds: 2
    lease-renewal-interval-in-seconds: 1
```

8.0 Test delle performance

Lo scopo di questa sezione è quella analizzare il comportamento della macchina su cui viene eseguita l'applicazione Eureka e verificare, a fronte dell'aumento progressivo delle richieste in entrata, quando si avrà un peggiormaneto delle prestazioni utilizzando le confgurazioni di default del Client Eureka.
Le misurazioni qui elencate sono una stima per quel che riguarda il Client Eureka, da precisare che questa relazione non è focalizzata sul piano di testing, in tal caso numerose misurazioni sarebbero necessarie per capire ancora meglio l'effettivo comportamento, soprattutto al cambiare dei parametri di configurazioni di Eureka.
Per la generazione e la misurazione delle richieste in entrata verrà utilizzato Jmeter.
Apache JMeter è un software open source sviluppato dall'Apache Software Foundation e liberamente scaricabile dal sito Web dedicato. Questo strumento non solo consente di mettere sotto carico le Web app, ma, con una serie di plugin e script sviluppati da terzi è possibile anche effettuare test sui database (tramite JDBC), sui server di posta (POP3 o IMAP), sui server Java Enterprise (JEE), sui server LDAP e altri.
Per valutare i risultati dei tst utilizzeremo dei componenti che mette a disposizione Jmeter chiamati Listener.
Un listener è un oggetto che permette il monitoraggio e report dei risultati delle richieste effettuate.
Vi sono diversi listener che è possibile utilizzare, in questo esempio verranno utilizzati principalmente 2 listener Summary Report e View Result in Tree.
Tali componenti permettono la misurazione del tempo medio di riposta per ogni pagina e inoltre offrono altre statistiche come il valore mediano, il valore minimo e massimo, la latenza, il Throughput, la linea a 90% ossia il tempo di risposta all'interno del quale cadono il 90% delle richieste e la percentuale degli eventuali errori presentati all' utente.
Per quanto riguarda le richieste da effettuare, i campi interessanti sono latenza, trhoughput e valore medio ri risposta, misureremo anche la percentulae di utilizzo di CPU e quantità di memria heap utilizzata dalla JVM sulla macchina target.
Per queste prove verranno utilizzate due macchine, una sarà usata solo per contenere una particolare applicazione che sarà poi l'oggetto del test.
La seconda macchina eseguirà le applicazioni web necessarie per rappresentare un tipico sistema distribuito con Eureka già elencato in precedenza.
Caratteristiche della macchina oggetto di test:

Nome Host 1:DESKTOP-O1LC34V
Processori:Quad core, Interl® Core™ i5-7500 CPU @ 3.40 GHz 3.40 GHz
RAM: 8GB
tipo HD:Samsung SSD 750 EVO 250GB
Sistema Operativo:Windows 10 Pro N 64bit

Nome Host 2:LAPTOP-U0QUUUFE
Processori: 3 Intel® Core™ i7-6500U CPU @ 2.50GHz 2.60GHz
RAM:8GB
tipo HD: WDC WD10JPVX-22JC3T0 1TB normale
Sistema Operativo:Windows 10 Home

8.1 Test di Eureka Client

Nella seguente tabella verranno elencati i risultati delle richieste effettuate al Client Eureka, applicazione incaricata di contattare il service registry periodicamente per reperire la lista aggiornata dei servizi e fornirli all'utente finale.

I parametri più rilevanti sono:

- Media:andamento del tempo medio di risposta espressa in millisecondi.
- Latenza:è la differenza tra il tempo di quando una richiesta viene inviata e quando viene ricevuta effettivamente dal server.
- Throughput: numero delle richieste processate per unità di tempo(secondi, minuti,ore) dal server.

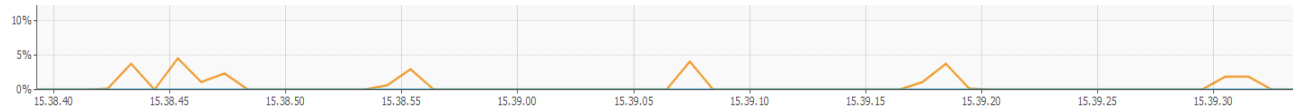
RICHIESTE LATO CLIENT

Per ogni quantità di richieste il suo risultato è espresso come media derivata dalla ripetizione di 5 esecuzioni.

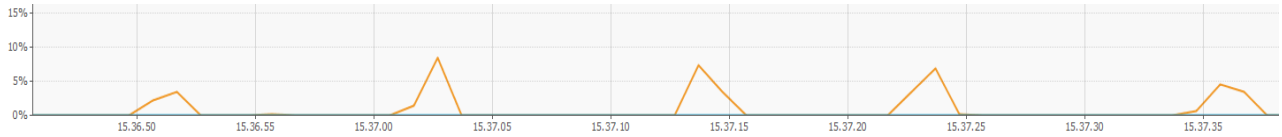
Qnt. Richieste	Latenza	Throughput	deviazione standard	media
100	9,242	90,965762	4,672	8,8
300	12,92	213,738338	16,402	12,4
600	17,53766667	326,383728	25,274	17,2
1000	33,7168	422,801992	70,526	33,4
1500	61,62146667	486,131338	94,716	61,2
2100	40,39333333	615,553624	87,464	40
2800	20,96321429	675,398378	25,786	20,4
3600	61,73466667	833,385016	69,83	61,2
4500	284,6702222	859,857308	269,932	284,2
5500	57,84807273	869,8070875	86,4575	50
6600	127,3976061	1078,222824	137,228	127,2
7800	143,4018205	1110,938488	164,582	143
9100	187,5006813	1137,498668	204,996	187
10500	267,4118857	1224,489336	268,846	267
12000	145,5642833	1275,077988	159,214	145
13600	244,4935147	1372,503528	325,334	244
15300	320,1193725	1389,480716	336,118	319,6
17100	180,3919649	1505,118238	171,484	179,8
19000	271,6337789	1412,098684	421,906	271,2
21000	219,4536667	1487,18604	187,848	219
23100	225,3066753	1521,659052	252,336	225
25300	256,787581	1647,662396	236,114	256,4
27600	265,3116812	1575,839736	308,056	264,8
30000	325,8411667	1635,463468	421,56	325,4
32500	409,2341477	1586,18294	565,866	409
35100	195,5400285	1746,939134	152,73	194,8
37800	253,6269101	1711,105278	360,564	253,2
40600	447,3533892	1942,033178	348,4325	368,25

UTILIZZO CPU LATO SERVER

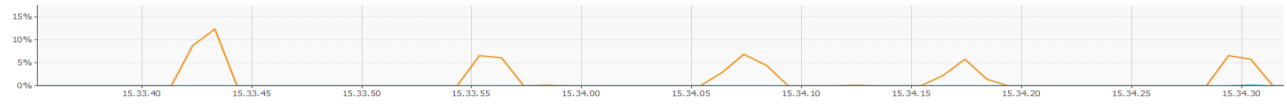
run 100



run 300

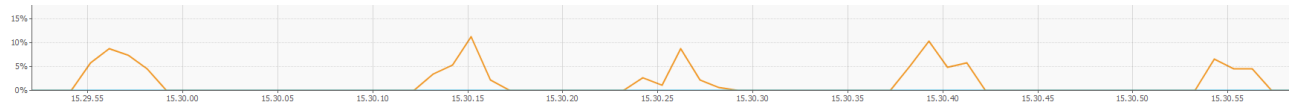


run 600

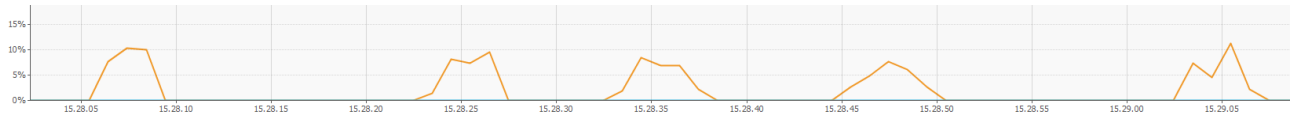


1000

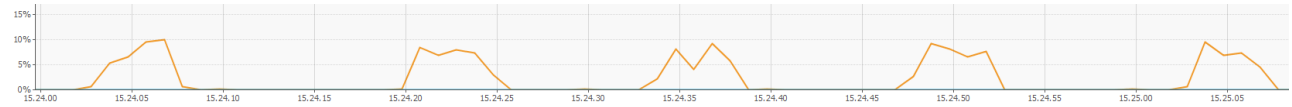
1500



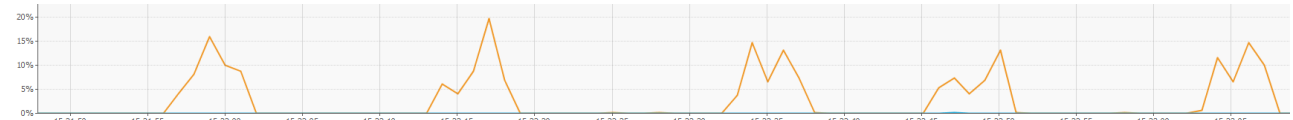
2100



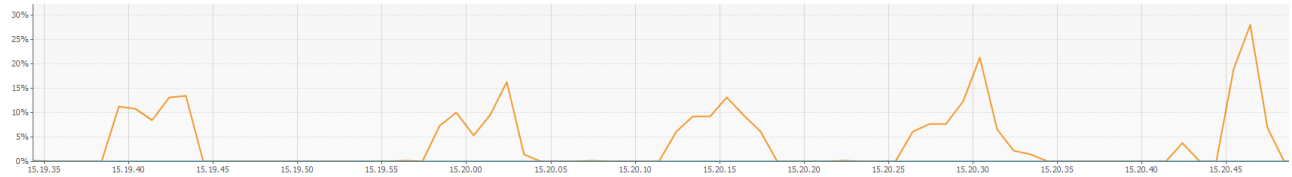
2800



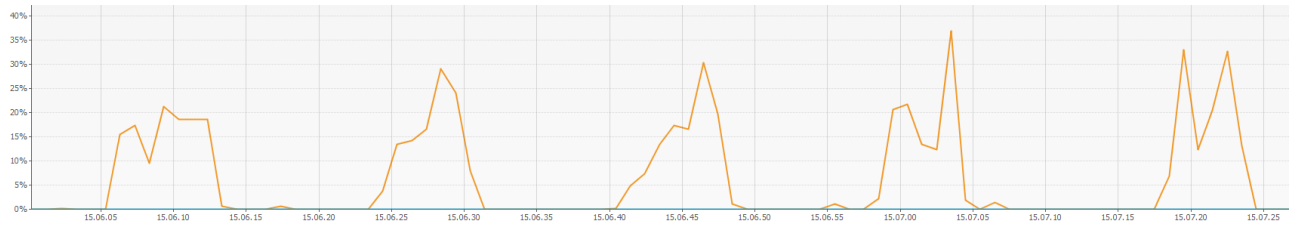
3600



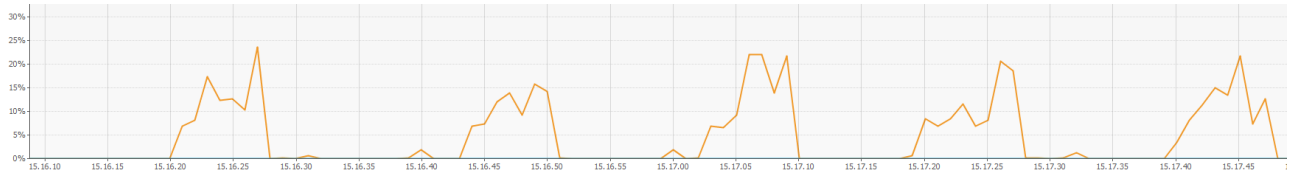
4500



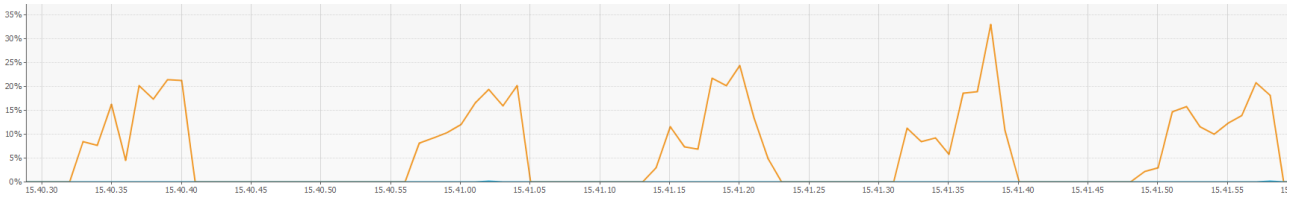
6600



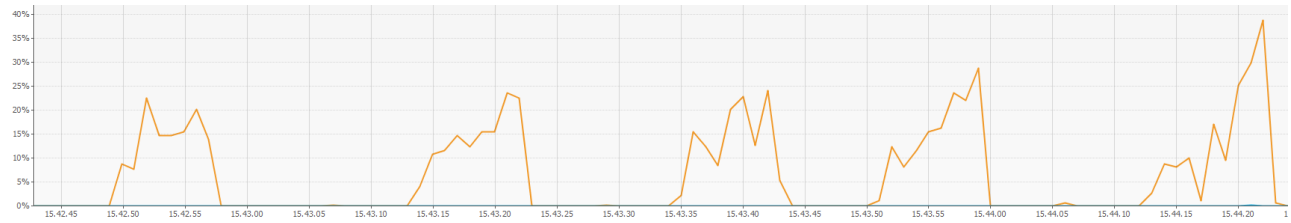
7800



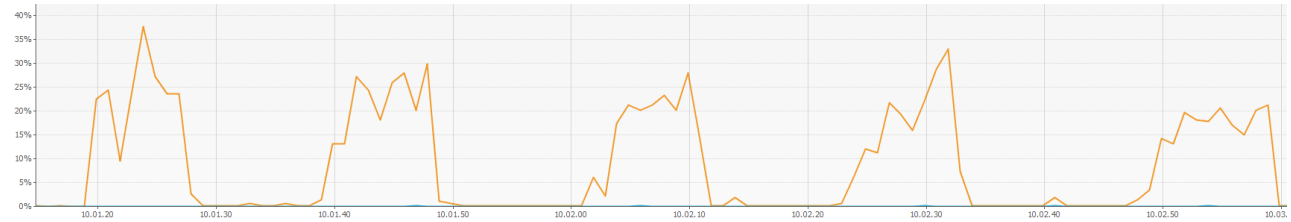
9100



10500

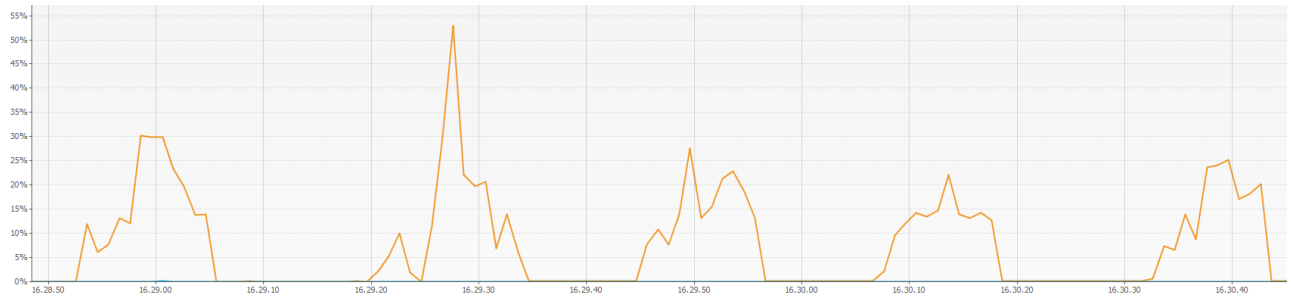


12000

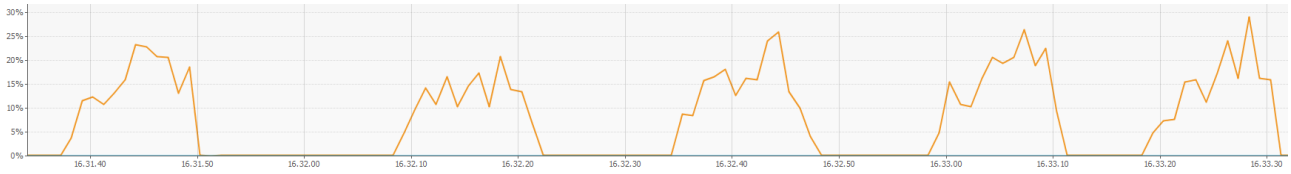


13600

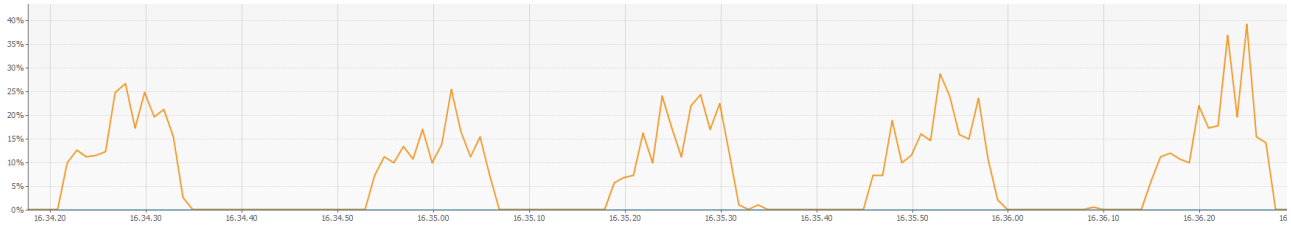
15300



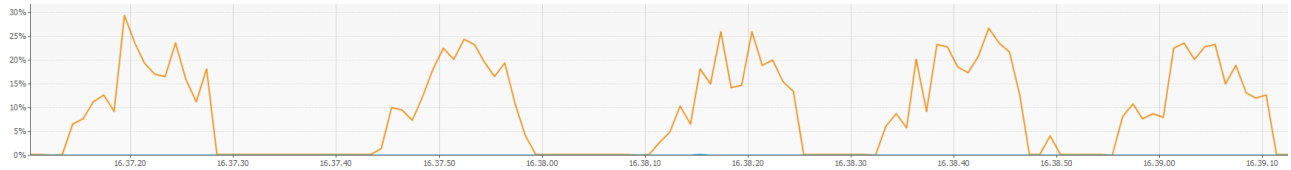
17100



19000

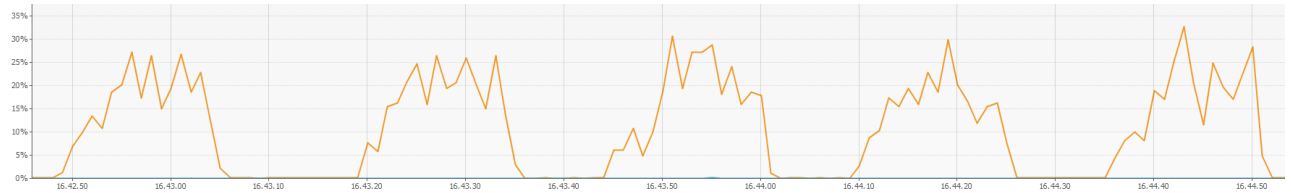


21000

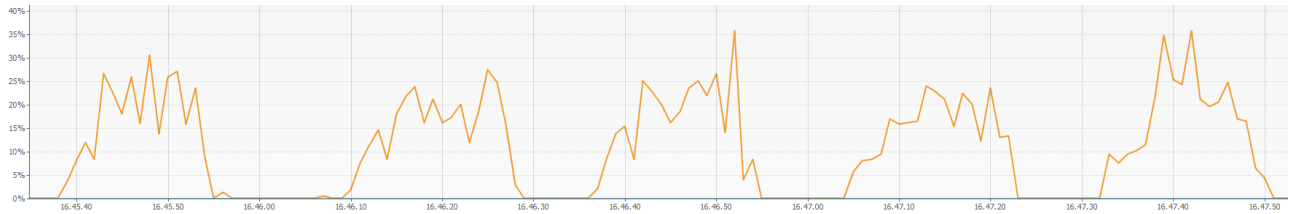


23100

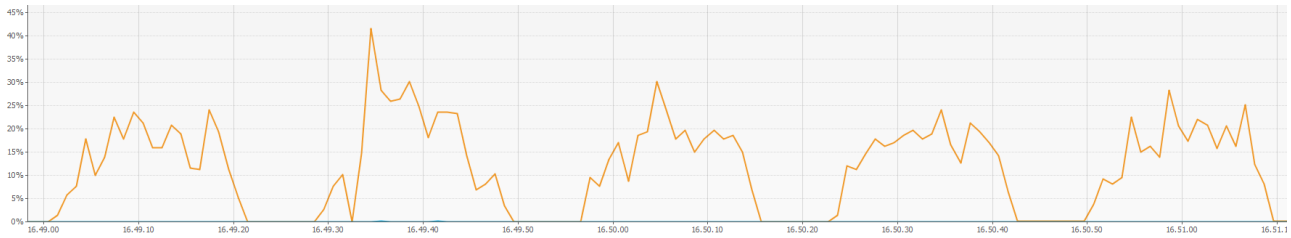
25300



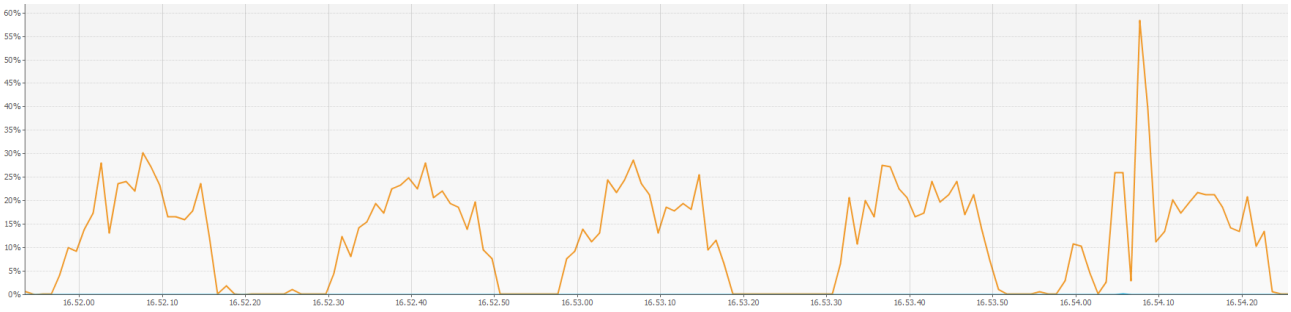
27600



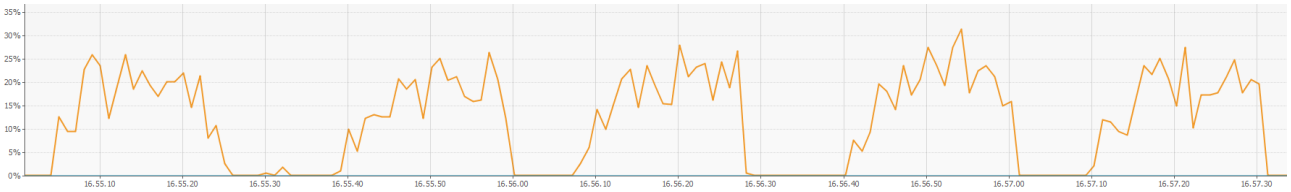
30000



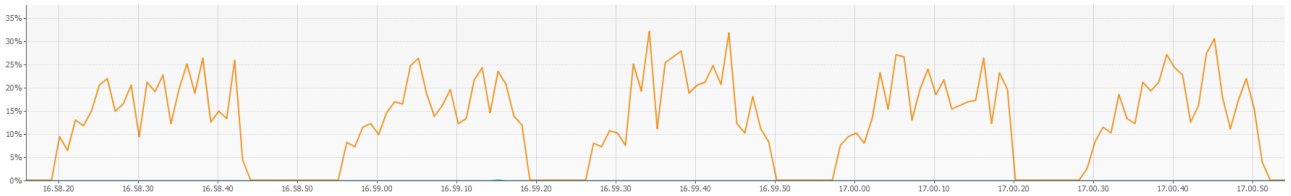
32500



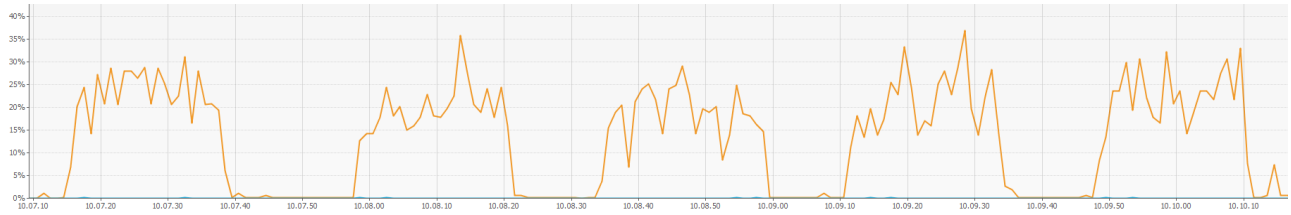
35100



37800

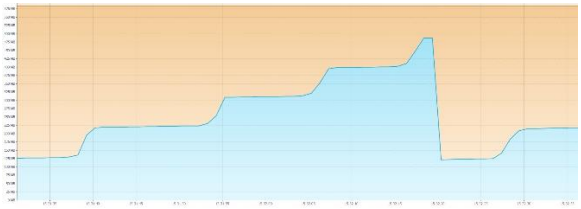


40600



Analizzando I grafici dellìutilizzo della CPU si puo notare che un primo peggioramento visisbile si ha oltrepassando la soglia delle 10500 richieste ma più marcato ancora il passaggioa oltre le 30k richieste.
Il periodo di processamento dele richieste da parte del Client Eureka diviene più lungo all'aumentare delle richieste e il conumo della CPU arriva piu volte a toccare la soglia del 35%.
I seguenti grafici invece descrivono il comportamento dell''heap della JVM.

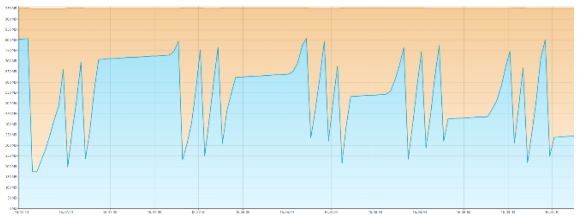
1000



6600

12000

21000



30000

Rispetto ai grafici sopra relativi alla CPU qui si ha un comportamente visibilmente piu marcato dell'uso della memoria.
Infatti si puo notare che i picchi sono più accentuati ma nmano all'0avicinarsi della soglia di 30k di richieste.
Se si ha necessità di aumentare la memoria disponibile nell'heap sarà necessario eseguire il programma con argomenti sul terminale o prompt dei comandi.

9.0 CONCLUSIONI

Al giorno d'oggi con il progressivo aumento dei dispositivi connessi in rete si ha l'esigenza di creare strutture per facilitare l'utente a trovare i servizi e oltre a questo a dare anche una garanzia sulla loro dispobinilità in qualsiasi momento.
Il servizio di discovery di Eureka viene gestito dal server registry il quale si preoccupa di conservare e aggiornare le informazioni sui servizi disponibili.
Il server registry gioca un ruolo fondamentale in Eureka, l'approccio centralizzato del registro potrebbe essere un aspeto critico ma grazie al servizio di caching fornito da Eureka e l'utilizzo di un bilanciatore del carico come Ribbon è possibile garantire un alta disponibilità del servizio e quindi rendere il sistema stabile a fronte di guasti e altri problemi.
L'approccio del bilanciamento del carico di tipo client side di Ribbon è utile per risolvere problemi di performance rispetto all'approccio server side.
Con un Client side load balancing si puo direttamente contattare il servizio desiderato attraverso un solo passaggio (dopo aver scoperto i servizi con il servizio di Discovery), a differenza dai tradizionali server side load balancing che ne richiedon o 2 evitando anche il collo di bottiglia per l'eventuale proxy incaricato a distribuire il carico.
Grazie alla tenologia Spring è possibile creare sistemi web based in modo semplificato evitando di perdere troppo tempo per i file di configurazione che di solito si usavano nei sistemi passati.
Netflix ha saputo integrare perfettamente le proprie tecnologie con Spring e a mio parere ha saputo anche adottare le strategie d ibuiseness giuste rilasciando le propre tecnologie con licenza opensource in modo da renderle utilizzabili da tutti e con possibilità di partecipare alla comunità proponendo miglioramenti.
Oltre a questo Spring ha la il vantaggio di poter esser integrato da inumerevoli tencologie e questo approccio modulare è fondamentale in una relatà in continuo sviluppo.

Autore ome articolo anno publicaizone

[PRA15]Prabhat gangwar, "Service Oriented Architecture", 04/08/2015
[ECL04]Eclipse Foundation, Eclipse Public License, 2004
[CHR15]Chris Richardson,"Microservice Architecture", http://microservices.io/patterns/client-side-discovery.html, 2015
[SEB15]Sebastián Peyrott,"An introduction to Microservices", https://auth0.com/blog ,2015
[ORL16]Orlando L Otero,"Spring Cloud Series", http://tech.asimio.net/2016/11/14/Microservices-Registration-and-Discovery-using-Spring-Cloud-Eureka-Ribbon-and-Feign.html, 2016
[EBE15]Eberhard Wolff, "Microservice with Spring Boot and Spring Cloud", https://www.slideshare.net/ewolff/microservice-with-spring-boot-and-spring-cloud, 2015
[DAV12]David Liu, "Netflix Eureka", https://github.com/Netflix/eureka/wiki, 2012
[BRE00] BREWER, Eric A.: Towards robust distributed systems, 2000