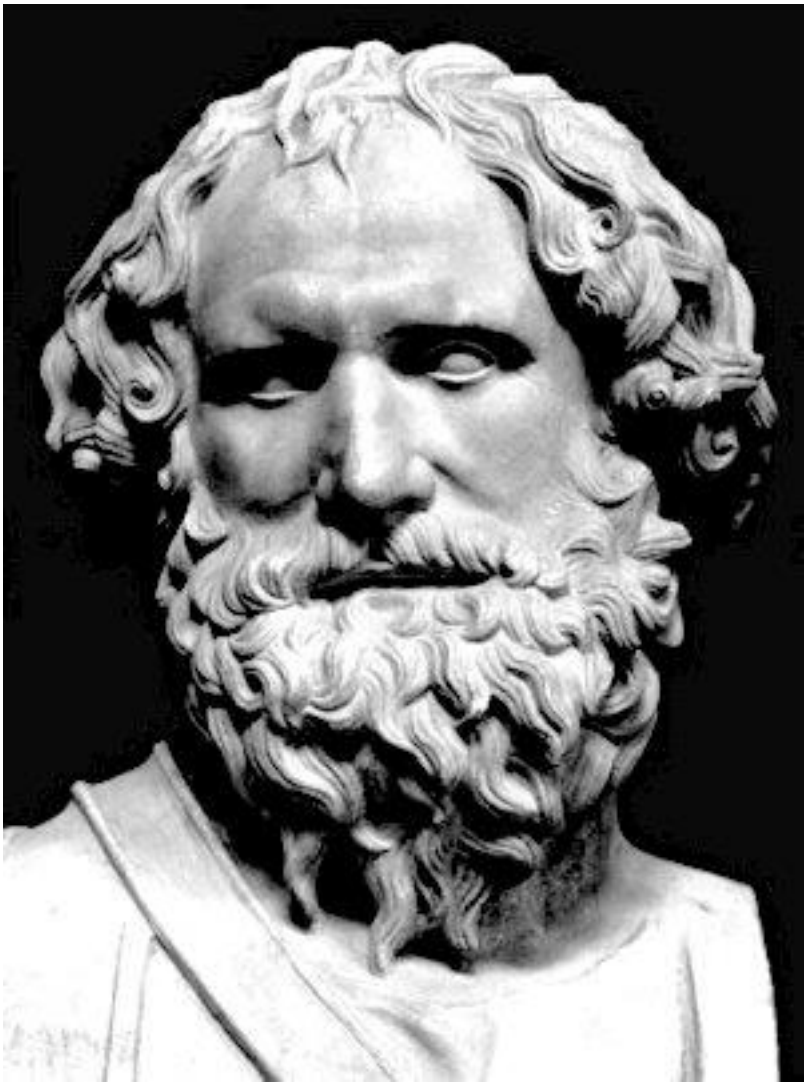


EUREKA

Reference Document

EUREKA NETFLIX OSS

What is Eureka? Know about the Netflix OSS Service Registry and Discovery Application

Shane Reddy

Table of Contents

1. Introduction

What is Eureka? Eureka is a Service Discovery software that was written purely in Java and open sourced by Netflix under their OSS (Open Source Software) suite.

By the advent of cloud technologies like elasticity where the running instances of the application or service are dynamic and provisioned and de-provisioned on the fly, service discovery where one service needs to find the other service is running is no easy matter.

What is service discovery means?

- Services have no prior knowledge about the physical location of other Service Instances.
- Services advertise their existence and disappearance.
- Services can find instances of another Service based on advertised metadata.
- Instance failures are detected, and they become invalid discovery results.
- Service Discovery is not a single point of failure by itself.

2. Overview

Netflix Eureka architecture consists of two components, the Server and the Client.

The Server is a standalone application and is responsible for:

- Managing a registry of Service Instances,
- Provide means to register, de-register and query Instances with the registry,
- Registry propagation to other Eureka Instances (Servers or Clients).

The Client is part of the Service Instance ecosystem and has responsibilities like:

- Register and unregister a Service Instance with Eureka Server,
- Keep alive the connection with Eureka Server,
- Retrieve and cache discovery information from the Eureka Server.

3. Tuning Parameters.

3.1 Client Settings

- ***getRegistryFetchIntervalSeconds (registryFetchIntervalSeconds)*** = Indicates how often (in seconds) to fetch the registry information from the eureka server.
- ***getEurekaServerReadTimeoutSeconds*** = Indicates how long to wait (in seconds) before a read from eureka server needs to timeout.
- ***getEurekaServerConnectTimeoutSeconds*** = Indicates how long to wait (in seconds) before a connection to eureka server needs to timeout.
- ***getEurekaServerTotalConnections*** = Gets the total number of connections that is allowed from eureka client to all eureka servers. Use ***getEurekaServerTotalConnectionsPerHost***" per host.
- ***shouldRegisterWithEureka (registerWithEureka)*** = Indicates whether this instance should register its information with eureka server for discovery by others. In some cases, you do not want your instances to be discovered whereas you just want to do discover other instances.
- ***shouldFetchRegistry (fetchRegistry)*** = Indicates whether this client should fetch eureka registry information from eureka server.
- ***shouldDisableDelta*** = Indicates whether the eureka client should disable fetching of delta and should rather resort to getting the full registry information. Note that the delta fetches can reduce the traffic tremendously, because the rate of change with the eureka server is normally much lower than the rate of fetches.
- ***getEurekaConnectionIdleTimeoutSeconds*** = Indicates how much time (in seconds) that the HTTP connections to eureka server can stay idle before it can be closed. It is recommended that the values are 30 seconds or less, since the firewall cleans up the connection information after a few mins leaving the connection hanging in limbo.

3.2 Server Settings

- ***eureka.instance.lease-renewal-interval-in-seconds*** = Indicates the frequency the client sends heartbeats to server to indicate that it is still alive. It’s not advisable to change this value since self-preservation assumes that heartbeats are always received at intervals of 30 seconds.

- ***eureka.instance.lease-expiration-duration-in-seconds*** = Indicates the duration the server waits since it received the last heartbeat before it can evict an instance from its registry. This value should be greater than lease-renewal-interval-in-seconds. Setting this value too small could make the system intolerable to temporary network glitches.
- ***eureka.server.eviction-interval-timer-in-ms*** = A scheduler is run at this frequency which will evict instances from the registry if the lease of the instances is expired as configured by lease-expiration-duration-in-seconds. Setting this value too long will delay the system entering self-preservation mode.
- ***actual number of heartbeats in last minute*** < ***expected number of heart beats per minute*** = enter self-preservation mode.
- ***eureka.server.responseCacheUpdateIntervalMs =*** The server maintains the cache that is refreshed every 30 seconds (default). The client also maintains the cache which is refreshed every 30 seconds. If the application uses Ribbon it also maintains cache which gets refreshed every 30 seconds.

3.3 Lease Settings

These decide the lease settings between the server and client as they determine the how aggressive you want to ping the server (hear beats) and tune this property to have better results during the network glitches.

Below is the excerpt taken directly from the Eureka ***LeaseInfo*** code:

```
public static final int DEFAULT_LEASE_RENEWAL_INTERVAL = 30;
public static final int DEFAULT_LEASE_DURATION = 90;

// Client settings
private int renewalIntervalInSecs = DEFAULT_LEASE_RENEWAL_INTERVAL;
private int durationInSecs = DEFAULT_LEASE_DURATION;

// Server populated
private long registrationTimestamp;
private long lastRenewalTimestamp;
private long evictionTimestamp;
private long serviceUpTimestamp;
```

The two important settings are below:

- ***DEFAULT_LEASE_DURATION*** = Sets the client specified setting for eviction (e.g. how long to wait without renewal event) time in seconds after which the lease would expire without renewal. This is set with "***leaseExpirationDurationInSeconds***" property.

- ***DEFAULT_LEASE_RENEWAL_INTERVAL =*** Sets the client specified setting for renew interval. The time interval with which the renewals will be renewed. This is set with **"*leaseRenewalIntervalInSeconds*"** property.

3.4 HTTP Settings

Below are the default HTTP connections settings:

- *DEFAULT_CONNECTION_IDLE_TIMEOUT = 30; (seconds)*
- *HTTP_CONNECTION_CLEANER_INTERVAL_MS = 30; (seconds)*
- *DEFAULT_MAX_CONNECTIONS_PER_HOST = 50;*
- *DEFAULT_MAX_TOTAL_CONNECTIONS = 200;*

4. Safeguard Features

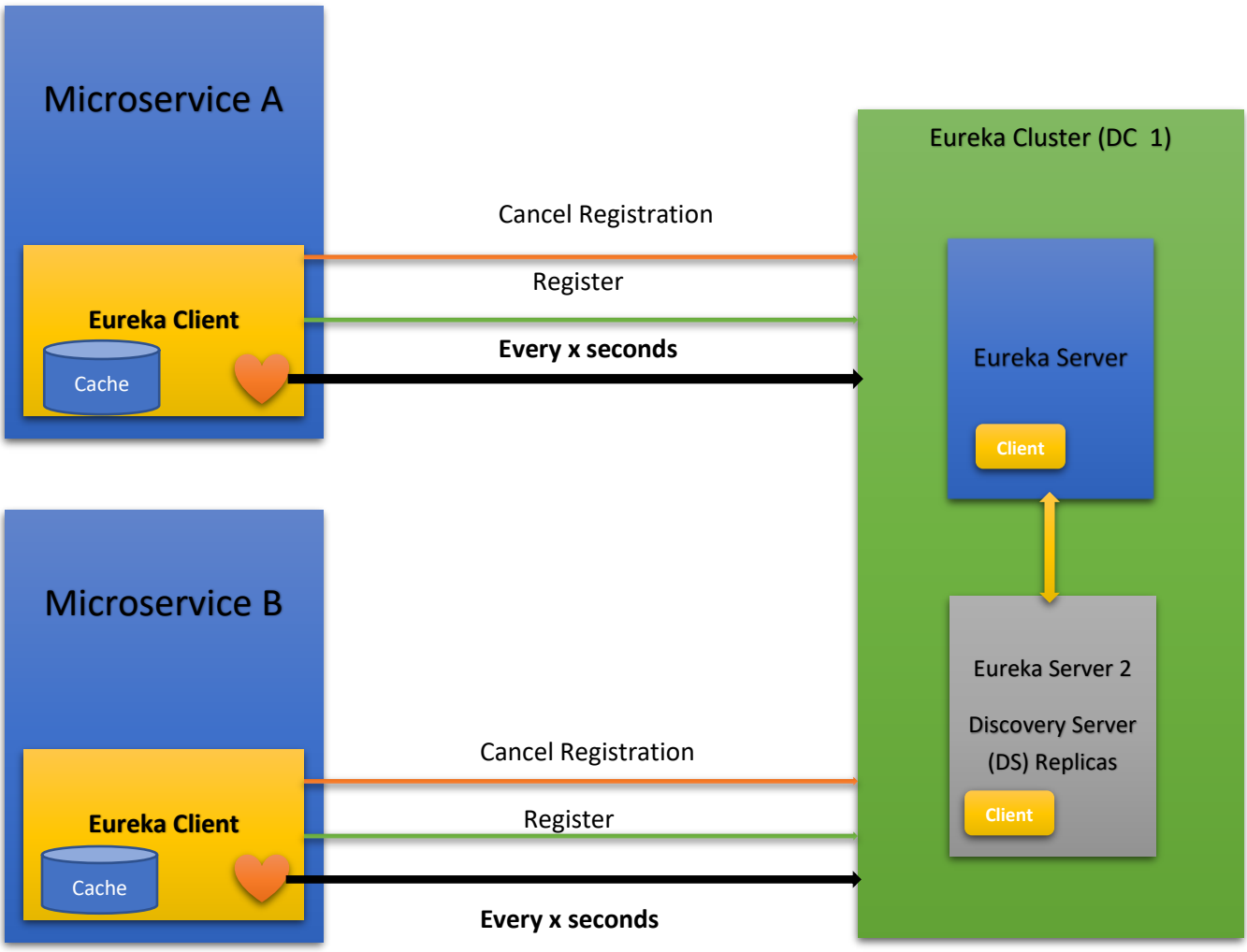
I think Eureka has two features which I would consider ***safeguard*** features, let's see what they are:

- Server self-preservation mode:
  In case a certain number of Instances fail to send heartbeats in a determined time interval, the Server will not remove them from the registry. It considers that a network partition occurred and will wait for these Instances to come back.

- Client-Side Caching:
  The Client pulls regularly discovery information from the registry and caches it locally. It basically has the same view on the system as the Server. In case all Servers go down or the Client is isolated from the Server by a

network partition, it can still behave properly until its cache becomes obsolete.

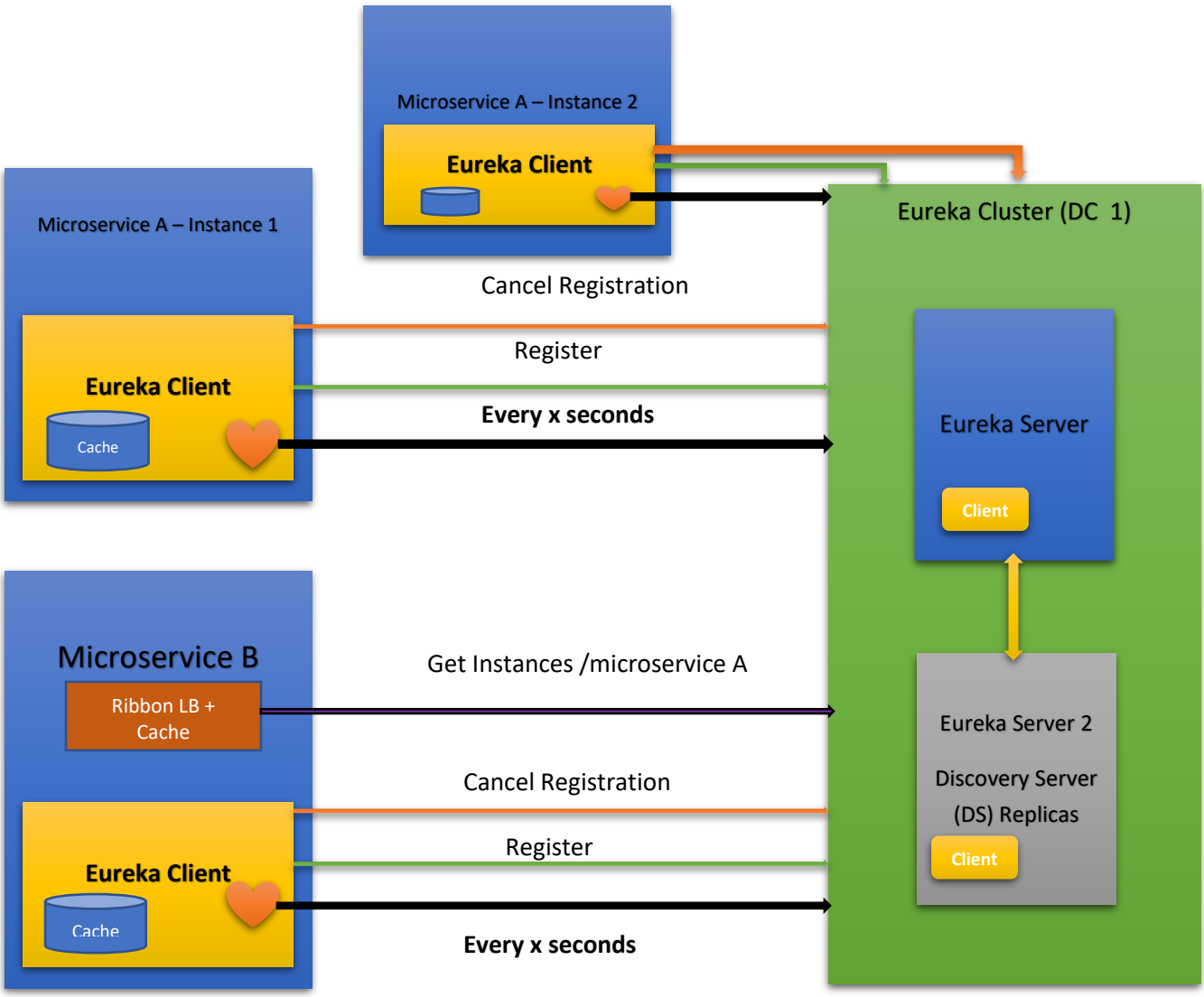
5. Registration and Discovery Topology



x= leaseRenewalIntervalInSeconds

Figure 1

6. Eureka with Ribbon Topology



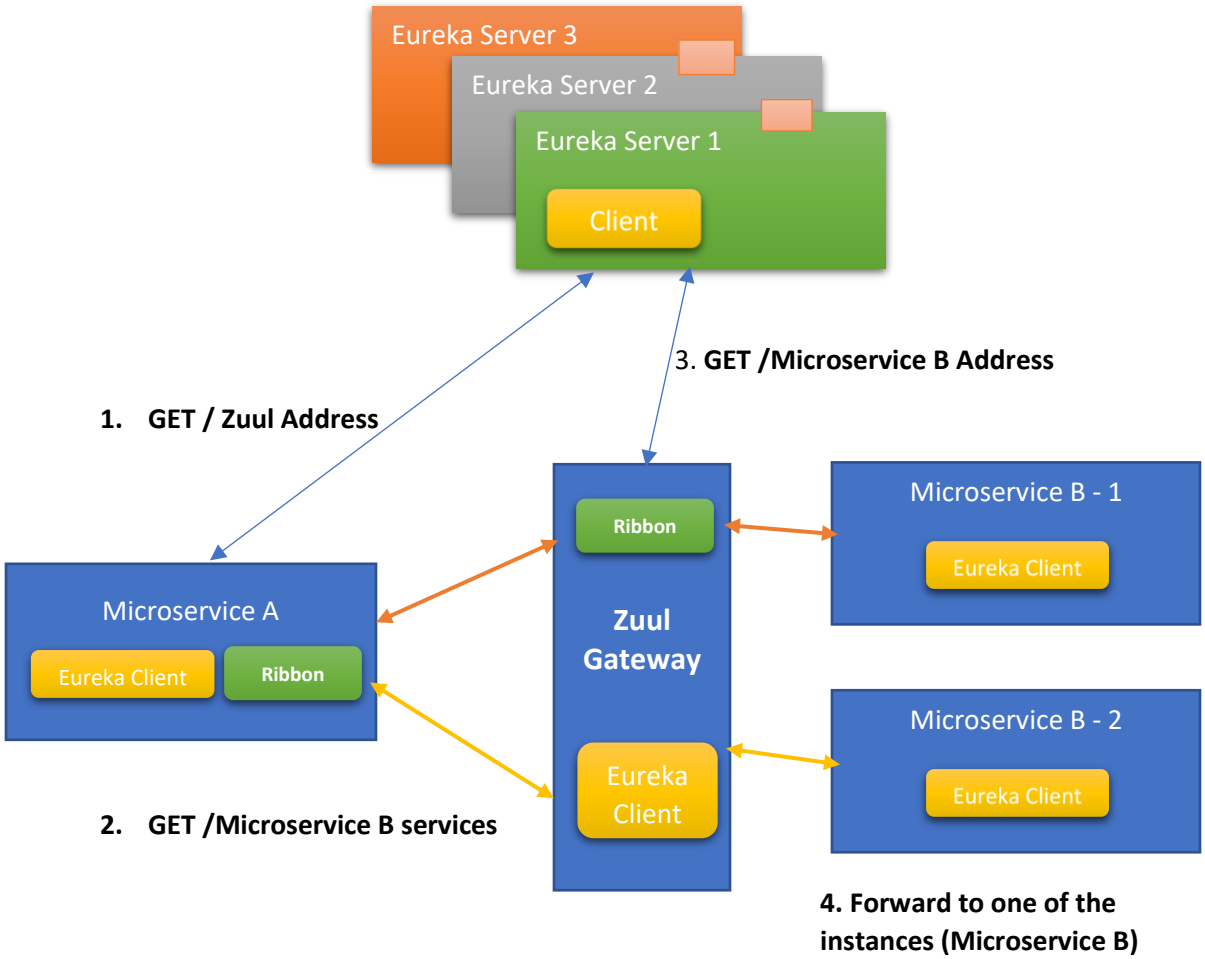
x= leaseRenewalIntervalInSeconds

Figure 2

In the above topology the Ribbon tool is used to load balance between the two instances of microservice A. Ribbon is part of Netflix OSS. The next graduation for this setup will be to use Zuul Gateway (Netflix OSS) to front the external connections.

7. Eureka + Ribbon + Zuul

Pre-requisite: All the services are registered with Eureka Cluster.



The microservice A needs a service that is hosted by microservice B which has 2 instances, the flow starts with A making the API call to Zuul Gateway for which it makes the connection to Eureka for Zuul address and then sends the API request to Zuul which will query Eureka for service B and load balances using Ribbon to one of the two instances.

Adding security to gateway will make the system more robust.