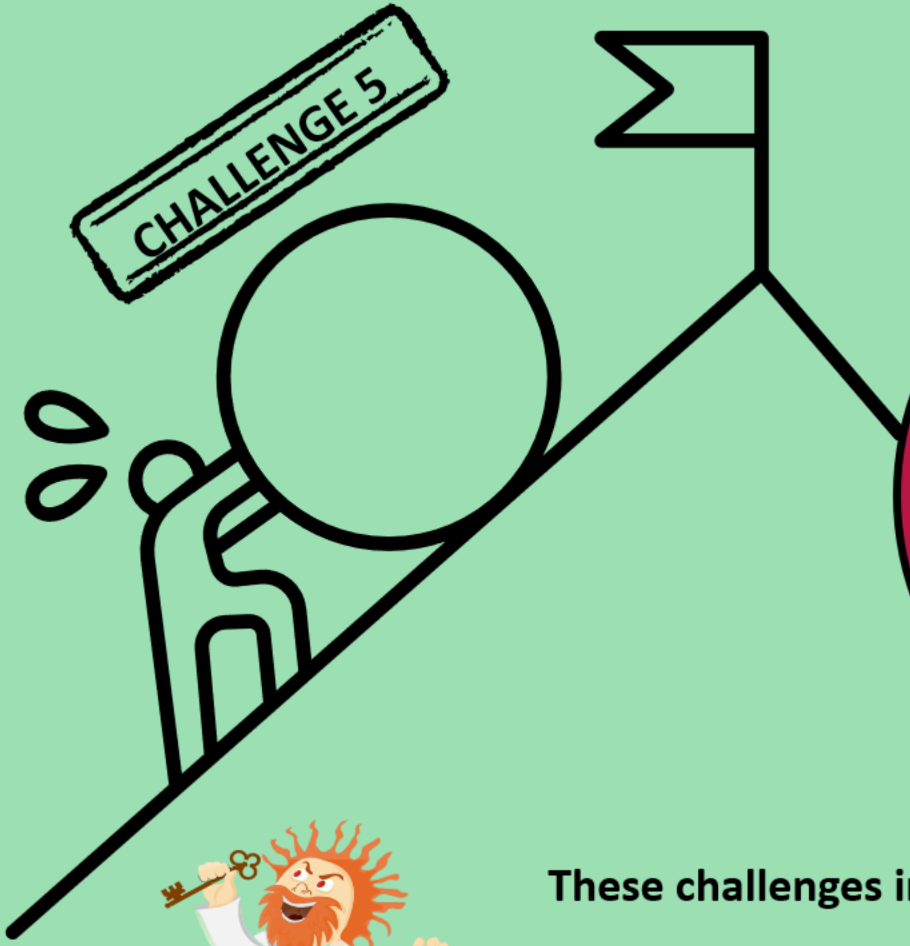
SERVICE DISCOVERY & REGISTRATION IN MICROSERVICES

eazy bytes



HOW DO SERVICES LOCATE EACH OTHER INSIDE A NETWORK?

Each instance of a microservice exposes a remote API with it's own host and port. how do other microservices & clients know about these dynamic endpoint URLs to invoke them. So where is my service?

HOW DO NEW SERVICE INSTANCES ENTER INTO THE NETWORK?

If an microservice instance fails, new instances will be brought online to ensure constant availability. This means that the IP addresses of the instances can be constantly changing. So how does these new instances can start serving to the clients?

LOAD BALANCE, INFO SHARING B/W MICROSERVICE INSTANCES

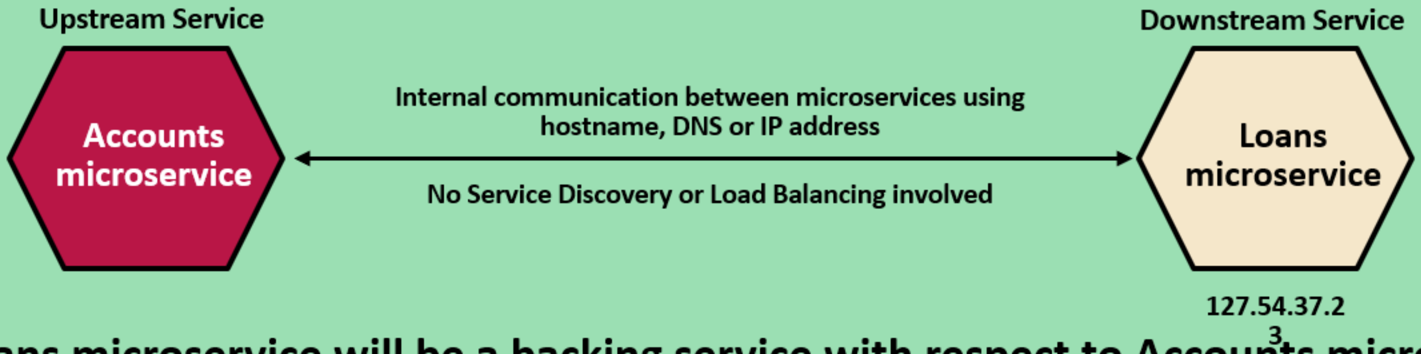
How do we make sure to properly load balance b/w the multiple microservice instances especially a microservice is invoking another microservice? How do a specific service information shared across the network?

These challenges in microservices can be solved using below concepts or solutions,

1) **Service discovery**
2) **Service registration**
3) **Load balancing**

How service communication happens in Traditional apps ?

Inside web network, When a service/app want to communicate with another service/app, it must be given the necessary information to locate it, such as an IP address or a DNS name. Let's examine the scenario of two services, Accounts and Loans. If there was only a single instance of Loans microservice, below figure illustrates how the communication between the two applications would occur.


Upstream Service
Accounts microservice
Internal communication between microservices using hostname, DNS or IP address
No Service Discovery or Load Balancing involved
Downstream Service
Loans microservice
127.54.37.23

Loans microservice will be a backing service with respect to Accounts microservice

When there is only one instance of the Loans microservice running, managing the DNS name and its corresponding IP address mapping is straightforward. However, in a cloud environment, it is common to deploy multiple instances of a service, with each instance having its own unique IP address.

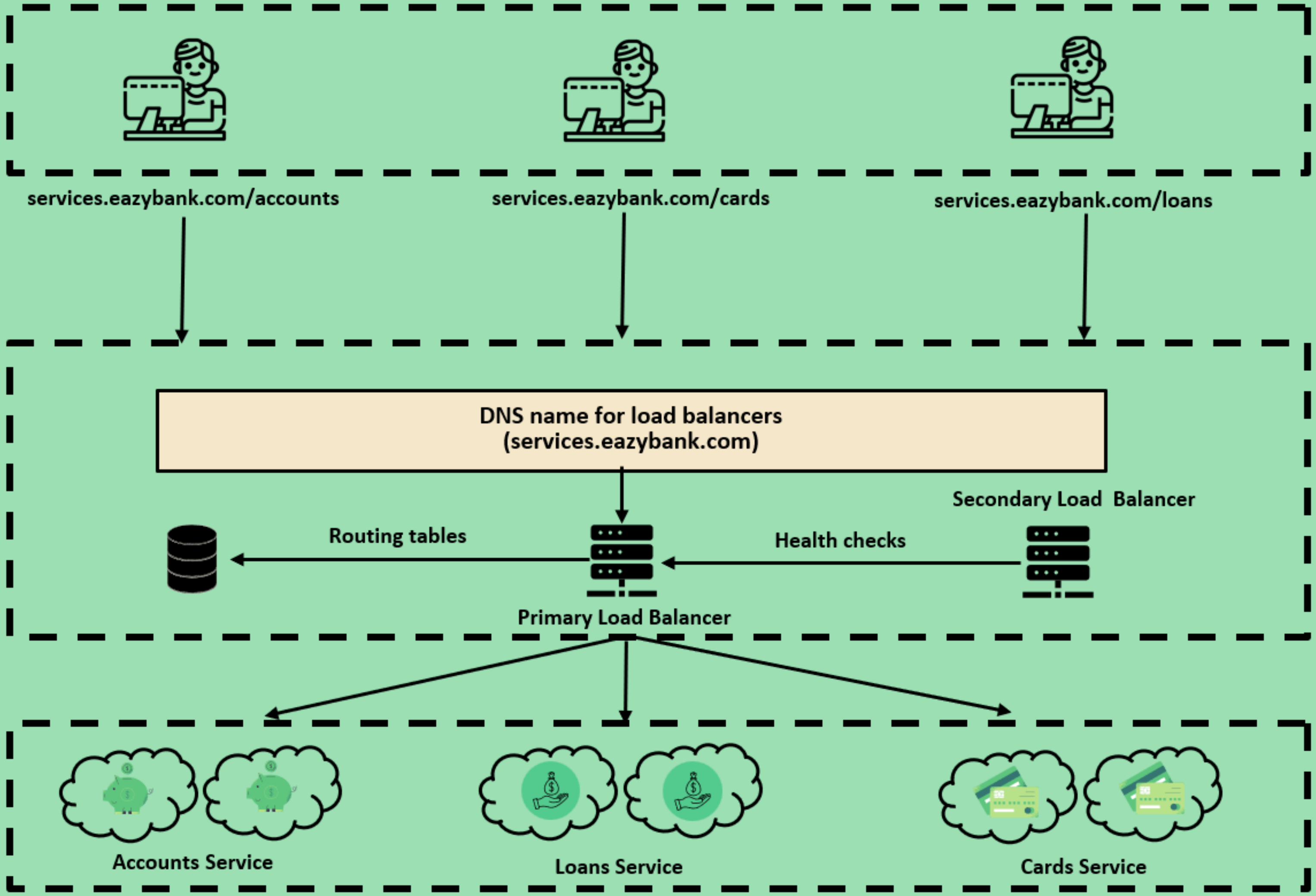
To address this challenge, one approach is to update DNS records with multiple IP addresses and rely on round-robin name resolution. This method directs requests to one of the IP addresses assigned to the service replicas in a rotating manner. However, this approach may not be suitable for microservices, as containers or services change frequently. This rapid change makes it difficult to maintain accurate DNS records and ensure efficient communication between microservices.

Unlike physical machines or long-running virtual machines, cloud-based service instances have shorter lifespans. These instances are designed to be disposable and can be terminated or replaced for various reasons, such as unresponsiveness. Furthermore, auto-scaling capabilities can be enabled to automatically adjust the number of application instances based on the workload.

How Traditional LoadBalancers works ?

eazy bytes

Clients like other services uses generic DNS along with the service specific path to invoke a specific service



Traditional Service location resolution architecture using DNS & a load balancer

Limitations with Traditional LoadBalancers ?

eazy bytes

With traditional approach each instance of a service used to be deployed in one or more application servers. The number of these application servers was often static and even in the case of restoration it would be restored to the same state with the same IP and other configurations.

While this type of model works well with monolithic and SOA based applications with a relatively small number of services running on a group of static servers, it doesn't work well for cloud-based microservice applications for the following reasons,

- Limited horizontal scalability & licenses costs
- Single point of failure & Centralized chokepoints
- Manually managed to update any IPs, configurations
- Complex in nature & not containers friendly

The biggest challenge with traditional load balancers is that some one has to manually maintain the routing tables which is an impossible task inside the microservices network. Because containers/services are ephemeral in nature

How to solve the problem for cloud native applications ?

eazy bytes

For cloud native applications, service discovery is the perfect solution. It involves tracking and storing information about all running service instances in a service registry.

Whenever a new instance is created, it should be registered in the registry, and when it is terminated, it should be appropriately removed automatically.

The registry acknowledges that multiple instances of the same application can be active simultaneously. When an application needs to communicate with a backing service, it performs a lookup in the registry to determine the IP address to connect to. If multiple instances are available, a load-balancing strategy is employed to evenly distribute the workload among them.

Client-side service discovery and server-side service discovery are distinct approaches that address the service discovery problem in different contexts

How to solve the problem for cloud native applications ?

eazy bytes

In a modern microservice architecture, knowing the right network location of an application is a much more complex problem for the clients as service instances might have dynamically assigned IP addresses. Moreover the number instances may vary due to autoscaling and failures.

Microservices service discovery & registration is a way for applications and microservices to locate each other on a network. This includes,

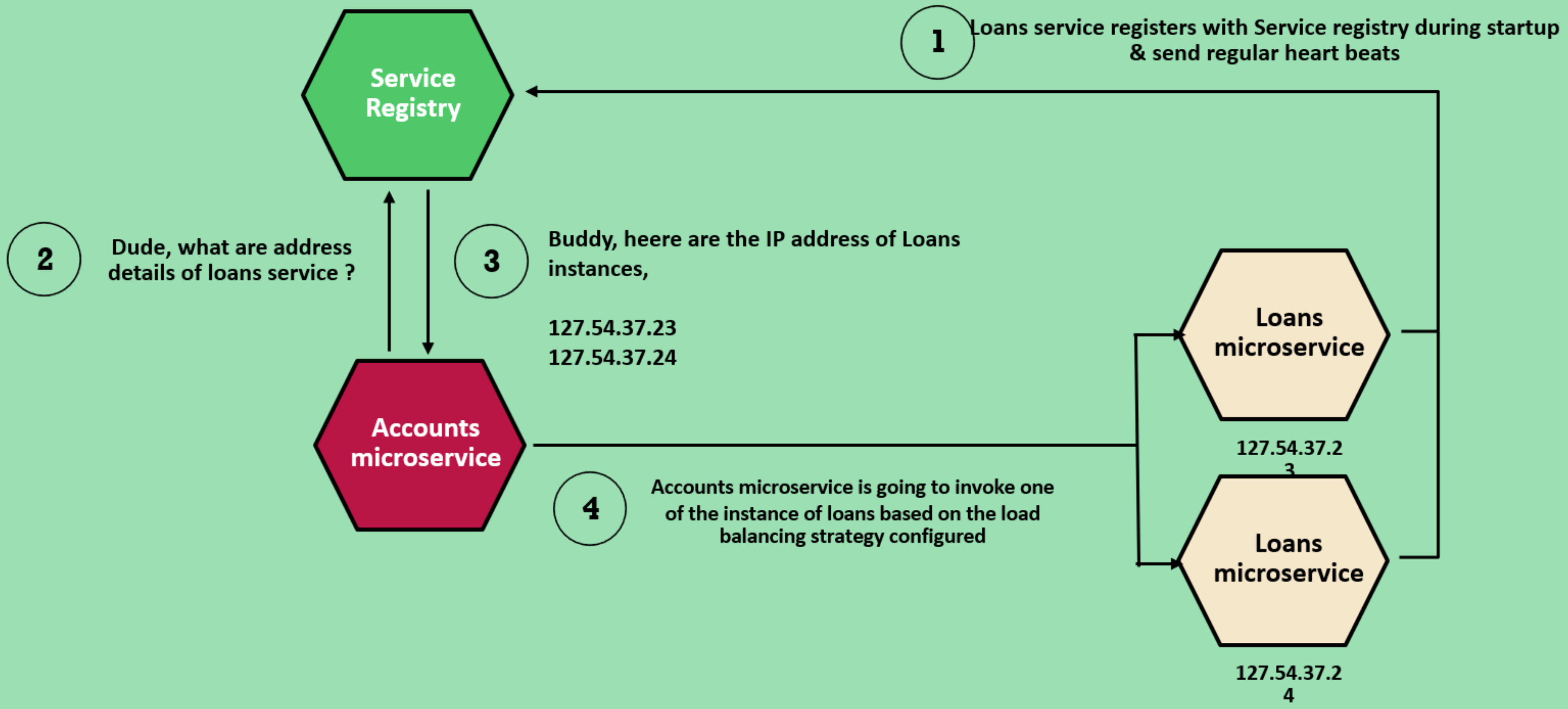
- A central server (or servers) that maintain a global view of addresses
- Microservices/clients that connect to the central server to register their address when they start & ready
- Microservices/clients need to send their heartbeats at regular intervals to central server about their health
- Microservices/clients that connect to the central server to deregister their address when they are about to shutdown

Service discovery & registrations deals with the problems about how microservices talk to each other, i.e. perform API calls.

Client-side service discovery and load balancing

eazy
bytes

In client-side service discovery, applications are responsible for registering themselves with a service registry during startup and unregistering when shutting down. When an application needs to communicate with a backing service, it queries the service registry for the associated IP address. If multiple instances of the service are available, the registry returns a list of IP addresses. The client application then selects one based on its own defined load-balancing strategy. Below figure illustrates the workflow of this process


Service Registry
1
Loans service registers with Service registry during startup & send regular heart beats
2
Dude, what are address details of loans service ?
3
Buddy, heere are the IP address of Loans instances,
127.54.37.23
127.54.37.24
Accounts microservice
Loans microservice
127.54.37.23
4
Accounts microservice is going to invoke one of the instance of loans based on the load balancing strategy configured
Loans microservice
127.54.37.24

Client-side service discovery and load balancing

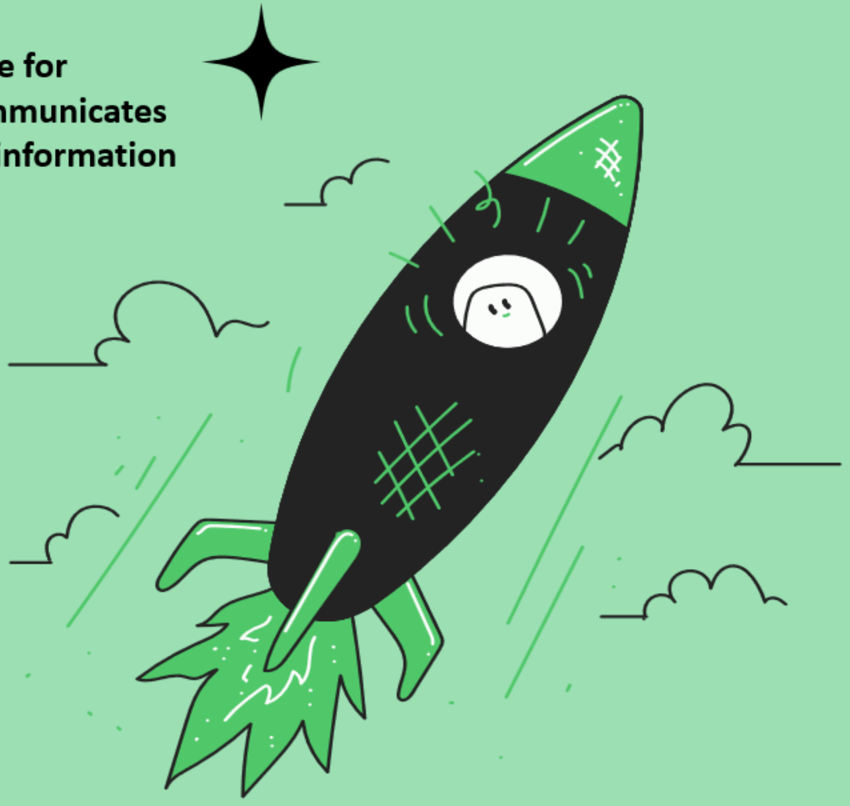
eazy bytes

Client-side service discovery is an architectural pattern where client applications are responsible for locating and connecting to services they depend on. In this approach, the client application communicates directly with a service registry to discover available service instances and obtain the necessary information to establish connections.

Here are the key aspects of client-side service discovery:

- **Service Registration:** Client applications register themselves with the service registry upon startup. They provide essential information about their location, such as IP address, port, and metadata, which helps identify and categorize the service.
- **Service Discovery:** When a client application needs to communicate with a specific service, it queries the service registry for available instances of that service. The registry responds with the necessary information, such as IP addresses and connection details.
- **Load Balancing:** Client-side service discovery often involves load balancing to distribute the workload across multiple service instances. The client application can implement a load-balancing strategy to select a specific instance based on factors like round-robin, weighted distribution, or latency.

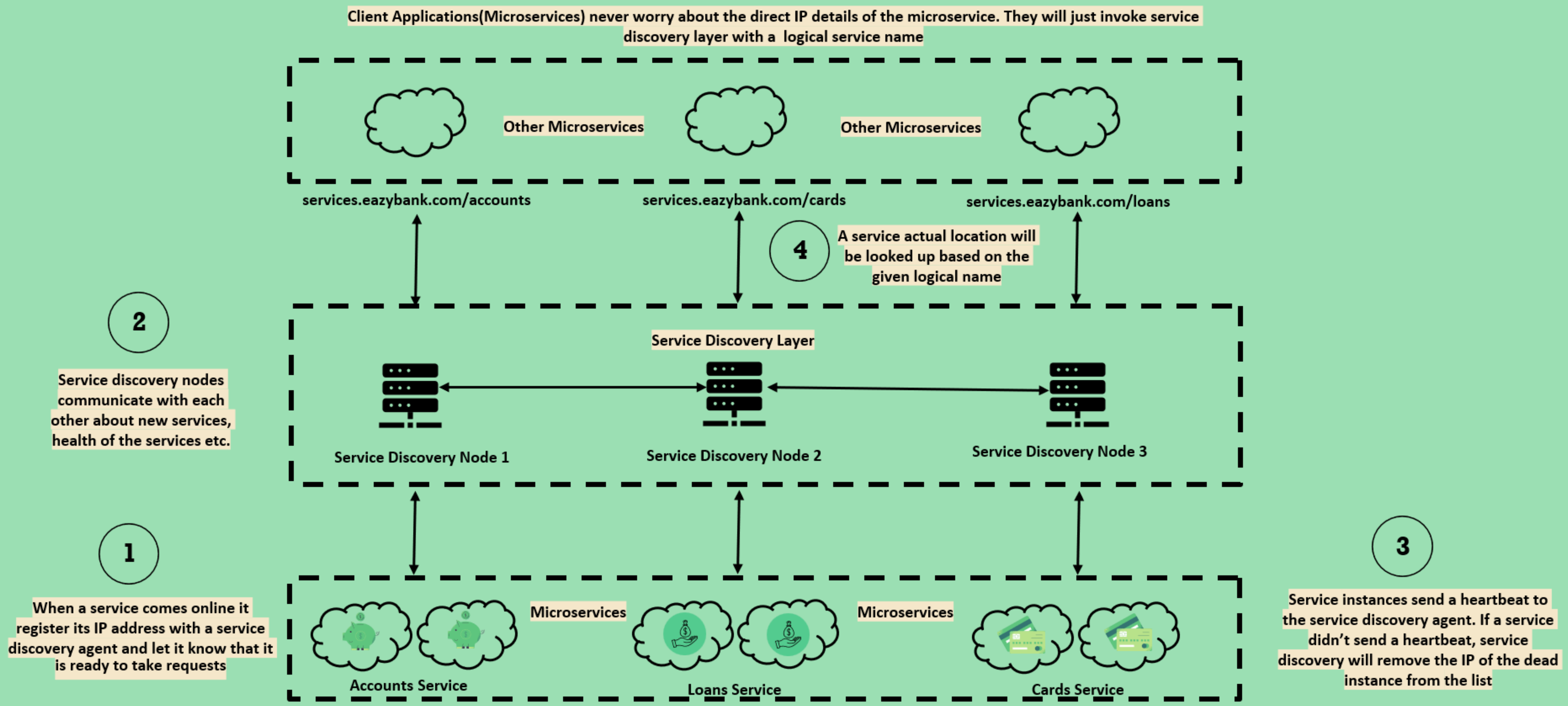
The major advantage of client-side service discovery is load balancing can be implemented using various algorithms, such as round-robin, weighted round-robin, least connections, or even custom algorithms. A drawback is that client service discovery assigns more responsibility to developers. Also, it results in one more service to deploy and maintain (the service registry). Server-side discovery solutions solve these issues. We are going to discuss the same when we are talking about Kubernetes



The Spring Cloud project provides several alternatives for incorporating client-side service discovery in our Spring Boot based microservices. More details to follow...

How Client-side service discovery works ?

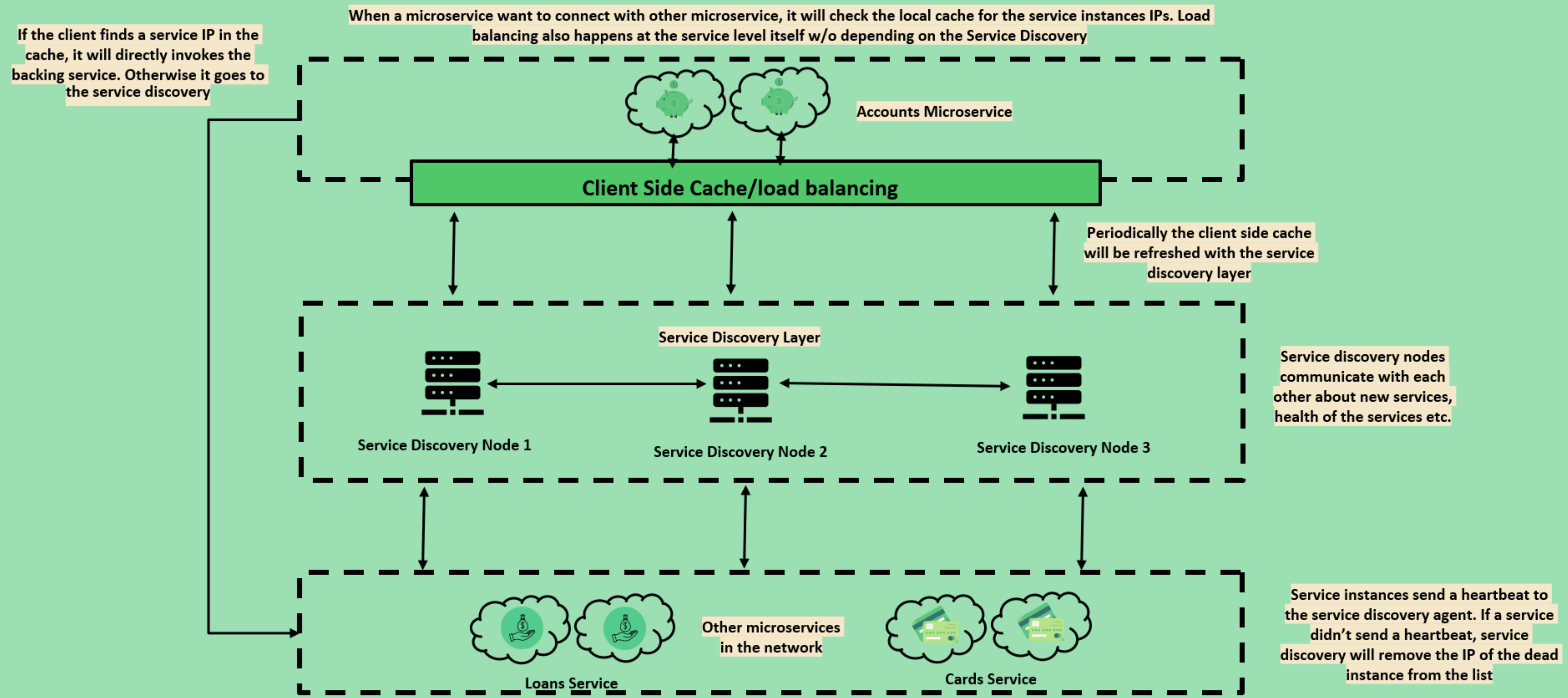
eazy bytes



Client Applications(Microservices) never worry about the direct IP details of the microservice. They will just invoke service discovery layer with a logical service name
Other Microservices
Other Microservices
services.eazybank.com/accounts
services.eazybank.com/cards
services.eazybank.com/loans
4
A service actual location will be looked up based on the given logical name
2
Service discovery nodes communicate with each other about new services, health of the services etc.
Service Discovery Layer
Service Discovery Node 1
Service Discovery Node 2
Service Discovery Node 3
1
When a service comes online it register its IP address with a service discovery agent and let it know that it is ready to take requests
Microservices
Microservices
Accounts Service
Loans Service
Cards Service
3
Service instances send a heartbeat to the service discovery agent. If a service didn't send a heartbeat, service discovery will remove the IP of the dead instance from the list

How loadbalancing works in Client-side service discovery ?

eazy bytes


If the client finds a service IP in the cache, it will directly invokes the backing service. Otherwise it goes to the service discovery
When a microservice want to connect with other microservice, it will check the local cache for the service instances IPs. Load balancing also happens at the service level itself w/o depending on the Service Discovery
Accounts Microservice
Client Side Cache/load balancing
Periodically the client side cache will be refreshed with the service discovery layer
Service Discovery Layer
Service Discovery Node 1
Service Discovery Node 2
Service Discovery Node 3
Service discovery nodes communicate with each other about new services, health of the services etc.
Other microservices in the network
Loans Service
Cards Service
Service instances send a heartbeat to the service discovery agent. If a service didn't send a heartbeat, service discovery will remove the IP of the dead instance from the list

Spring Cloud support for Client-side service discovery

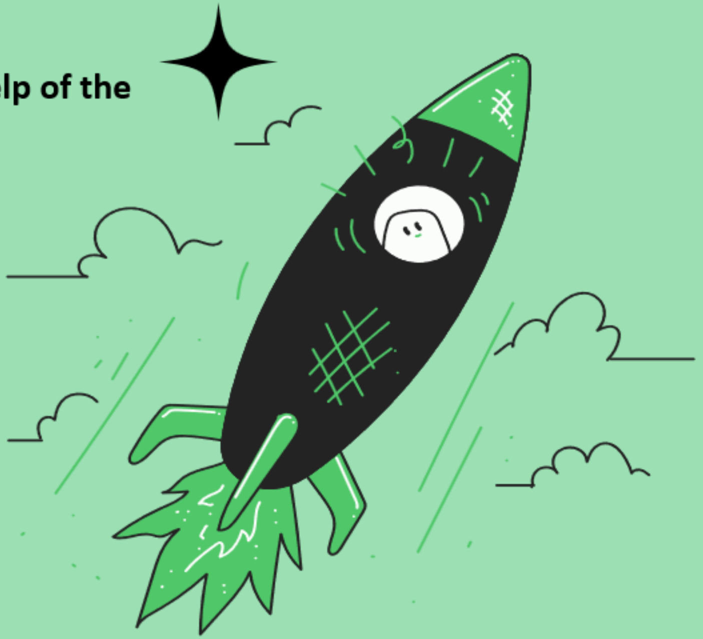
eazy bytes

Spring Cloud project makes Service Discovery & Registration setup trivial to undertake with the help of the below components,

- **Spring Cloud Netflix's Eureka** service which will act as a service discovery agent
- **Spring Cloud Load Balancer** library for client-side load balancing
- **Netflix Feign client** to look up for a service b/w microservices

Though in this course we use Eureka since it is mostly used but they are other service registries such as etcd,Consul, and Apache Zookeeper which are also good.

Though Netflix Ribbon client-side is also good and stable product, we are going to use Spring Cloud Load Balancer for client-side load balancing. This is because Ribbon has entered a maintenance mode and unfortunately, it will not be developed anymore



Advantages of Service Discovery approach includes,

- No limitations on availability
- Peer to peer communication b/w Services Discovery agents
- Dynamically managed IPs, configurations & Load balanced
- Fault-tolerant & Resilient in nature

Steps to build Eureka Server

Below are the steps to build a Eureka Server application using Spring Cloud Netflix's Eureka,

1. **Set up a new Spring Boot project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-netflix-eureka-server maven dependency.

2. **Configure the properties:** In the application properties or YAML file, add the following configurations,

```
server:
  port: 8070

eureka:
  instance:
    hostname: localhost
  client:
    fetchRegistry: false
    registerWithEureka: false
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

3. **Add the Eureka Server annotation:** In the main class of your project, annotate it with @EnableEurekaServer. This annotation configures the application to act as a Eureka Server.

4. **Build and run the Eureka Server:** Build your project and run it as a Spring Boot application. Open a web browser and navigate to http://localhost:8070. You should see the Eureka Server dashboard, which displays information about registered service instances.

Steps to register a microservice as a Eureka Client

eazy bytes

Below are the steps to make a microservice application to register and act as a Eureka client,

1. **Set up a new Spring Boot project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-netflix-eureka-client maven dependency.

2. **Configure the properties:** In the application properties or YAML file, add the following configurations,

```
eureka:
  instance:
    preferIpAddress: true
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: "http://localhost:8070/eureka/"
```

3. **Build and run the application:** Build your project and run it as a Spring Boot application. Open a web browser and navigate to http://localhost:8070. You should see the microservice registered itself as an application and the same can be confirmed inside the Eureka Server dashboard.

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes

In a distributed system using Eureka, each service instance periodically sends a heartbeat signal to the Eureka server to indicate that it is still alive and functioning. If the Eureka server does not receive a heartbeat from a service instance within a certain timeframe, it assumes that the instance has become unresponsive or has crashed. In normal scenarios, this behavior helps the Eureka server maintain an up-to-date view of the registered service instances.

However, in certain situations, network glitches or temporary system delays may cause the Eureka server to miss a few heartbeats, leading to false expiration of service instances. This can result in unnecessary evictions of healthy service instances from the registry, causing instability and disruption in the system.

To mitigate this issue, Eureka enters into Self-Preservation mode. When Self-Preservation mode is active, the existing registry entries will not be removed even if it stops receiving heartbeats from some of the service instances. This prevents the Eureka server from evicting all the instances due to temporary network glitches or delays.

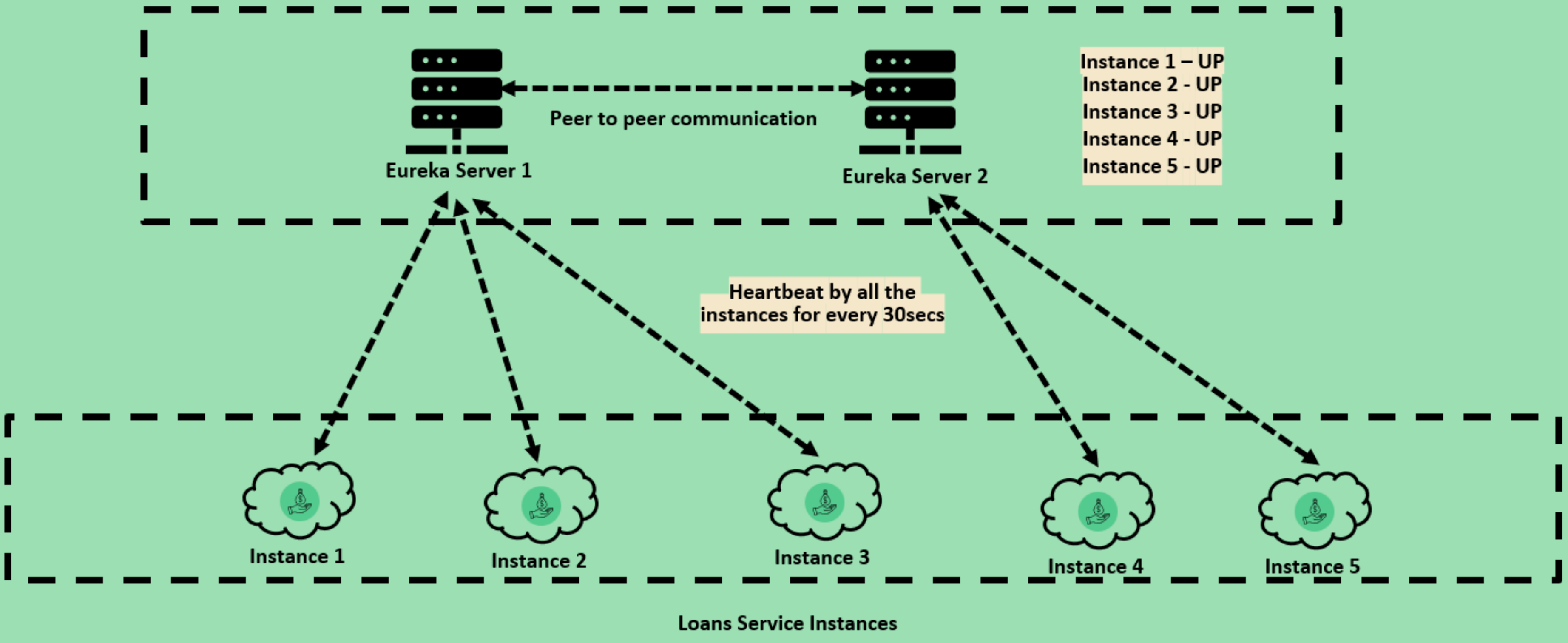
In Self-Preservation mode, the Eureka server continues to serve the registered instances to client applications, even if it suspects that some instances are no longer available. This helps maintain the stability and availability of the service registry, ensuring that clients can still discover and interact with the available instances.

Self-preservation mode never expires, until and unless the down microservices are brought back or the network glitch is resolved. This is because eureka will not expire the instances till it is above the threshold limit.

Eureka Server will not panic when it is not receiving heartbeats from majority of the instances, instead it will be calm and enters into Self-preservation mode. This feature is a savior where the networks glitches are common and help us to handle false-positive alarms.

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

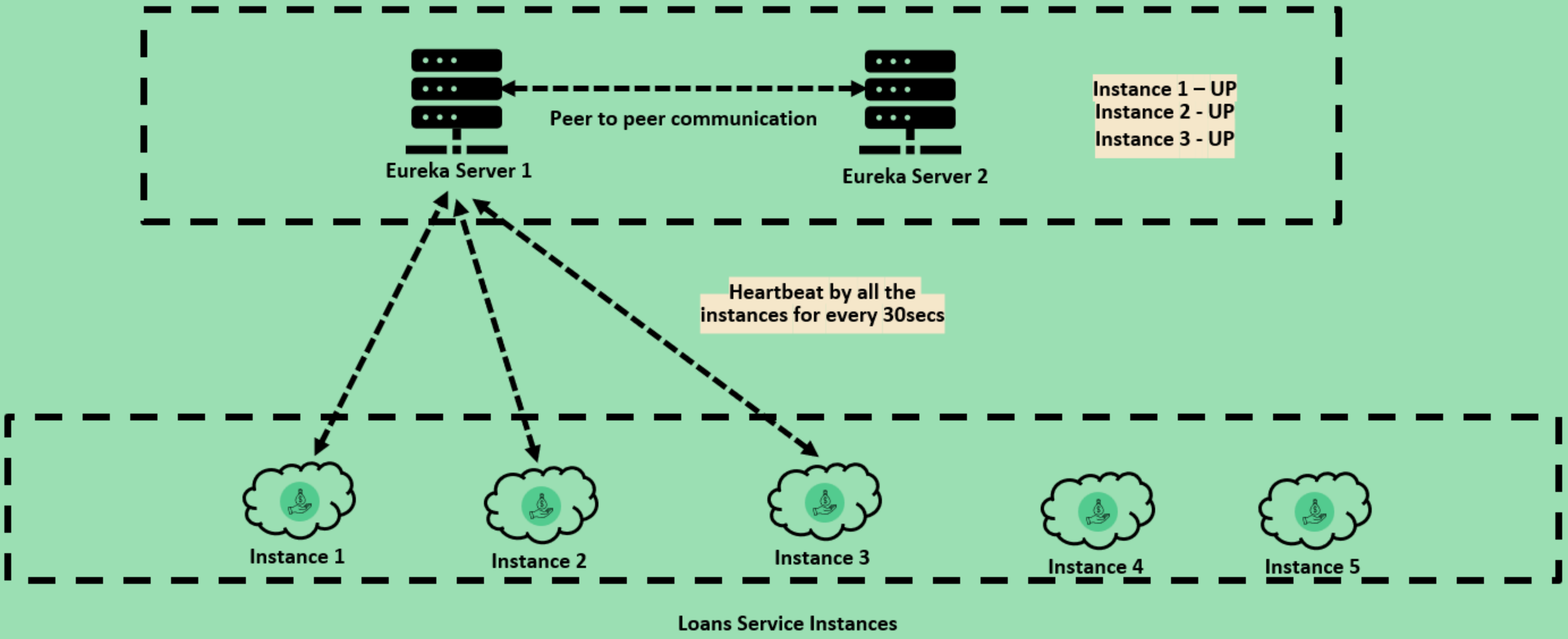
eazy bytes



Healthy Microservices System with all 5 instances up before encountering network problems

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

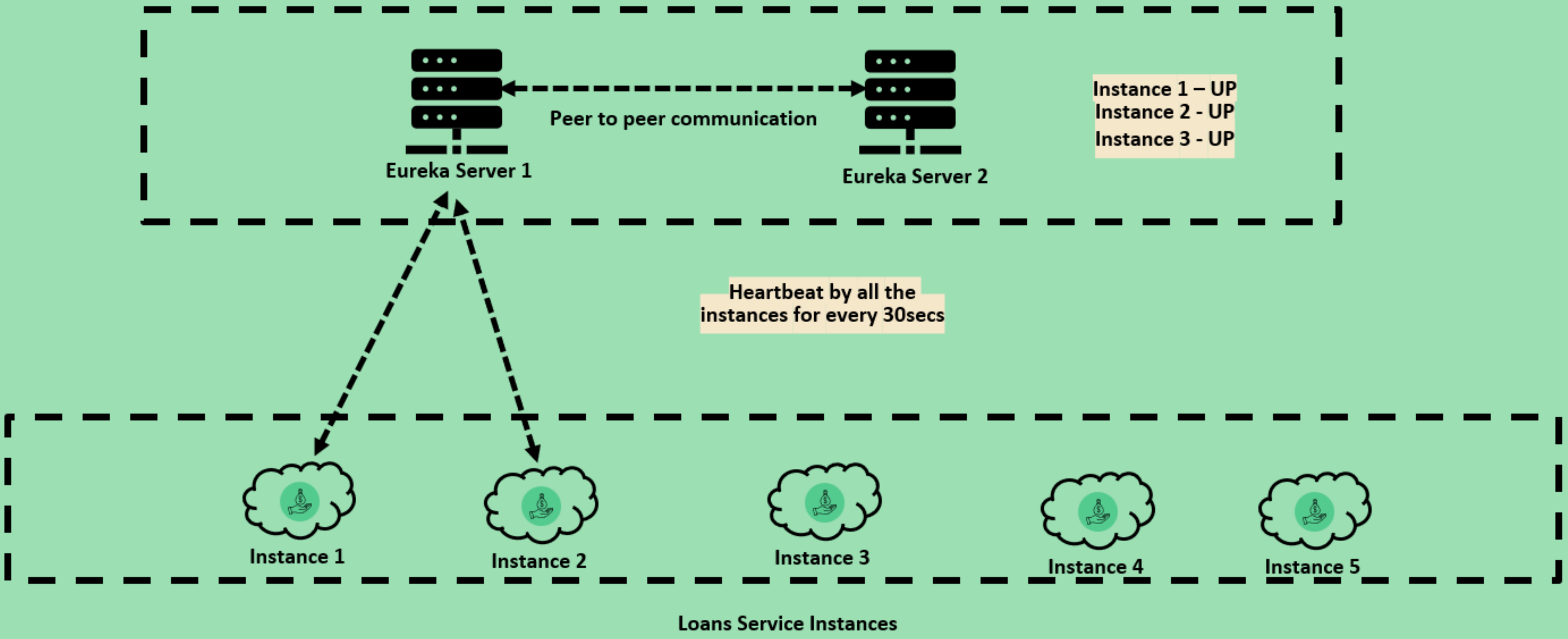
eazy bytes



2 of the instances not sending heartbeat. Eureka enters self-preservation mode since it met threshold percentage

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes



During Self-preservation, eureka will stop expiring the instances though it is not receiving heartbeat from instance 3

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes

- Configurations which will directly or indirectly impact self-preservation behavior of eureka

  ✓ `eureka.instance.lease-renewal-interval-in-seconds = 30`
  Indicates the frequency the client sends heartbeats to server to indicate that it is still alive

  ✓ `eureka.instance.lease-expiration-duration-in-seconds = 90`
  Indicates the duration the server waits since it received the last heartbeat before it can evict an instance

  ✓ `eureka.server.eviction-interval-timer-in-ms = 60 * 1000`
  A scheduler(EvictionTask) is run at this frequency which will evict instances from the registry if the lease of the instances are expired as configured by lease-expiration-duration-in-seconds. It will also check whether the system has reached self-preservation mode (by comparing actual and expected heartbeats) before evicting.

  ✓ `eureka.server.renewal-percent-threshold = 0.85`
  This value is used to calculate the expected % of heartbeats per minute eureka is expecting.

  ✓ `eureka.server.renewal-threshold-update-interval-ms = 15 * 60 * 1000`
  A scheduler is run at this frequency which calculates the expected heartbeats per minute

  ✓ `eureka.server.enable-self-preservation = true`
  By default self-preservation mode is enabled but if you need to disable it you can change it to ‘false’