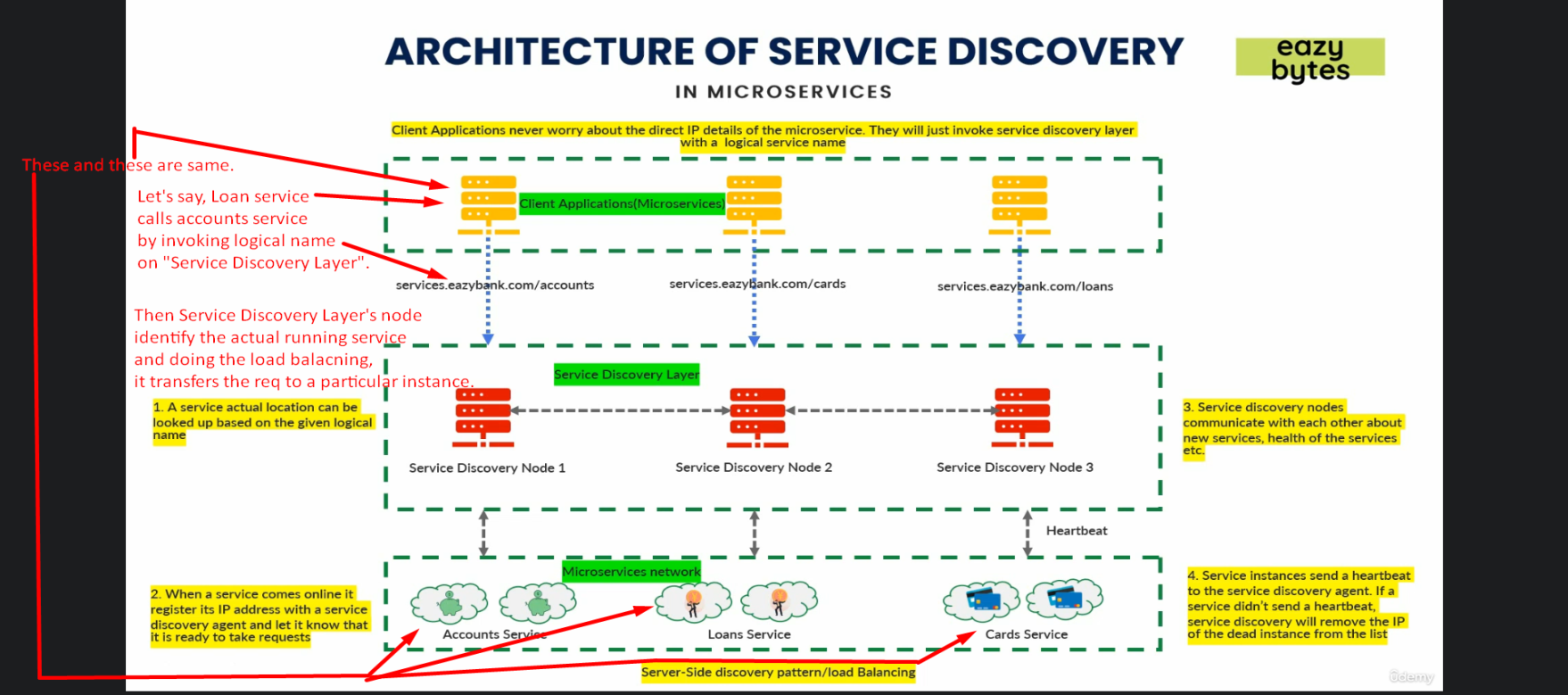
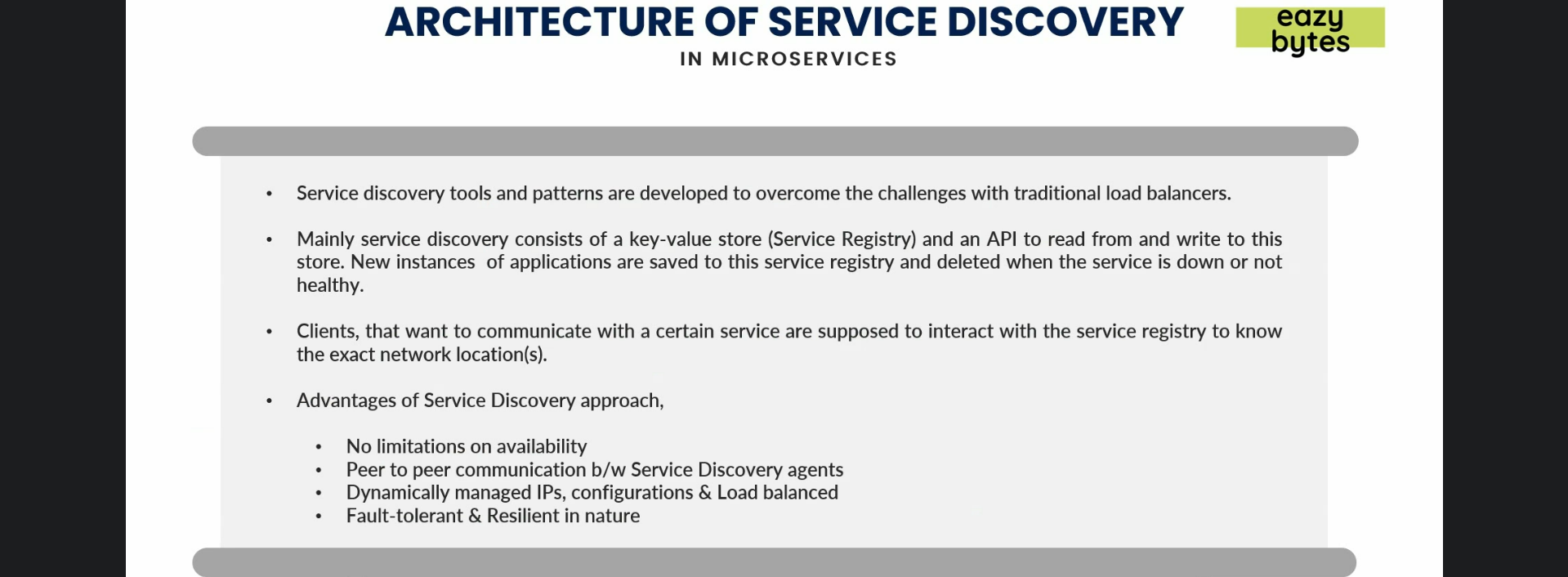
1. We discussed that how **“Service Discovery and Registration”** helps in the followings
   1. One microservice **to identify** another microservice in a network.
   2. How one microservice **registers** itself with it and comes into network topology so that other microservice can reach out to it.
   3. **How load balance** and info sharing b/w registered microservices happen.
2. But traditional load balancer doesn’t help to solve the above.
3. **Agenda**:
   1. How **“Service Discovery and Registration”** architecture works internally.
4. With minimum effort, we can achieve this architecture by using **Spring Cloud Project support**.
5. Let’s start with “**Service Discovery**” layer which is the **middle layer**.  
   The following is the architecture of **“Service Discovery”**.



* 1. As a developer, I’ve to make sure that I’m building the “**Service Discovery**” layer which is middle layer with **service discovery agent/node**.
  2. As per my requirement, if we need 3 nodes/agents of service discovery,
     1. Just like Spring Cloud Config Server, we have to make another microservice for each agent/node.
  3. This service discovery layer maintains all the address details and info about these microservices (account, loans) in our network.  
     So, this is very similar to Config Server which maintains environment configurations whereas Service Discovery agent will maintain the network locations of MSs and info.
  4. So, first we will build a microservice for our service discovery layer and then we start it which will act as service discovery node/agent.
  5. Once it is started, then we start our microservices (accounts, loans, cards) with multiple instances as a 2nd step.
     1. As soon as we start them, they go and register themselves with Service Discovery node.  
        They actually register their addresses with node along with logical names.
  6. Service discovery also expects a heartbeat from each registered microservice after each interval.  
     **Default time is 30 seconds**.  
     If a microservice instance doesn’t send signal within 30 seconds (maybe due to network issue, memory issue or slowness), it waits for next 30 and next 30 so total 90 seconds, eventually, it removes the registered details for that particular instance from the **registry**. Obviously, the reason is if some other microservice invokes that no-working microservice, it will get some exception or invalid response or timeout.
  7. If we have 3 service discovery nodes/agents, then microservice instance needs to send heartbeat to just one node not to all.  
     Because of **peer-to-peer communication**, one node transfers the received heartbeat to next node.  
     The way they communicate with each other is called “**Gossip/infection Protocol**“.
  8. **NOTE**: This architecture is for client app (Microservices in your network).  
     I mean suppose we have three services 🡺 accounts, loans, cards.   
     Now microservice accounts wants to communicate with microservice loans.   
     **This is not for some external service (UI, other service).  
     For UI we have API Gateway**.

1. Timeline

   Description automatically generated
2. So, this “Service Discovery and Registration” pattern solves all the following problems:
   1. **How a microservice knows about other microservice location.**  
      **Answer**: Through “**Service Discovery Layer**”
   2. **How one microservice registers itself and enters into the network where other microservices can interact with it**.  
      **Answer**: At the startup time of microservice, it will go and get itself registered with **“Service Discovery Layer”** and after each 30 seconds, it sends heartbeat signal to a Service Discovery node in the layer.
   3. **How a load balancing happens b/w multiple instances of a single microservice.**  
      **Answer:** The Service Discovery Layer itself will take care of load balancing by following some round-robin or other algo that you configure.
3. Service Discovery Tools and Patterns are developed to overcome the challenges that we faced in traditional load balance.
4. 
   1. Mainly service discovery consists of a registry which will be kind of key-value store and there will be an API which is exposed so that someone can invoke it to reveal information (read) and write information.
5. Advantages of “Service Discovery Layer”.
   1. **No limitations on availability**: You can add as many Service Discovery Nodes as you want to maintain load.
   2. **Pear-to-Pear communication b/w Service Discovery Agents**: With **Infection/Gossip protocol**, each node knows
      1. The health info about each microservice instance.
      2. The registration details about each instance.
   3. **Dynamically managed IPs, configurations & Load Balanced**: Nothing explained here.
   4. **Fault-tolerant & Resilient in nature**: With heartbeat signal, Service Discovery layer knows about which microservice instance is not working and it immediately deletes its registration details from Service Registry.
6. The load balancing is happening at the server level so we call it **Server level load balancing**.  
   It means every time a client (one microservice) tries to interact with the microservice network, it has to interact with the service discovery layer to identify what are the direct IP details and the **service discovery layer does the load balancing**.
7. We also have a **Client Side Load balancing** concept where my individual client app will take care of load balancing by itself without depending on service discovery layer and with client side load balancing we make sure that we’re not depending on the service discovery layer to a great extent but to some extent still we have to depend such as to fetch other microservice’s network details from.   
   **NOTE**: Client means other microservice in the same network. Not outside UI app or some other backend app outside of the organization.
8. Let’s discuss what client side load balancing is in next lecture.