CS7NS6 ASSIGNMENT

Exercise 2: Implementing a Globally-accessible Distributed Service

Group 7

Student(s) Name:

Prabhjot Kaur, 18335316

Karan Bharadwaj, 18335156

Shivani Tomar, 20303198

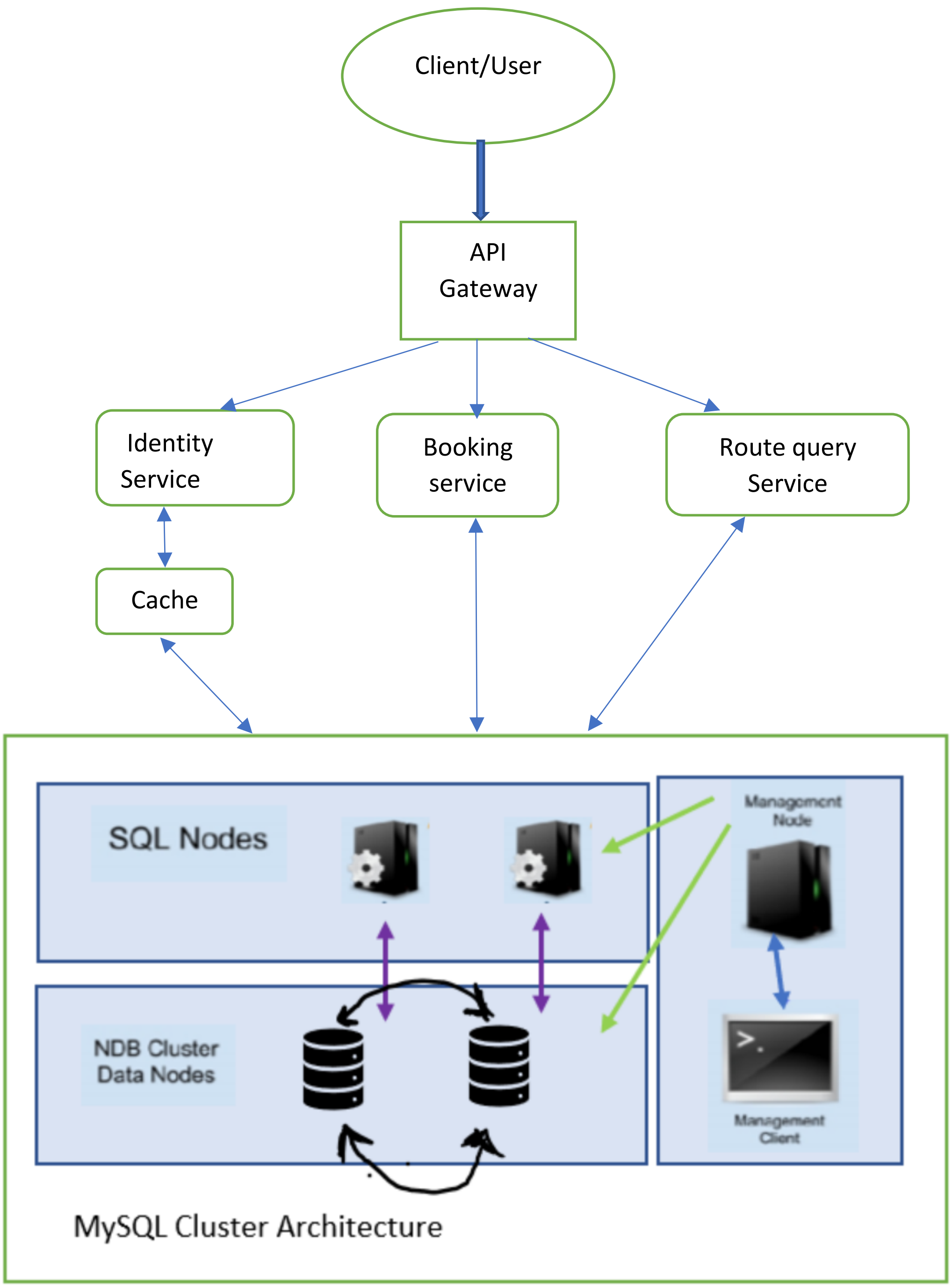
Rupasmita Devi, 20300052

Contents

Chapter 1: Introduction

We are implementing a globally-accessible journey booking distributed service which allows a user to start a journey only if a booking ticket is available. We have designed the system as a microservices based architecture. We have used various techniques such as atomic transactions, data replication, message logging, caching, load balancing in our attempt to make a highly reliable, fault tolerant and scalable service.

Chapter 2: Architecture:



The architectural components in the above diagram are described below:

API Gateway: This component will be responsible to forward the request to the appropriate instance of the application using load balancing algorithms.

Identity Service: Identity service is an individual microservice deployed for doing user authentication.

Route Query Service: Query Service is our another microservice currently used for querying route details for a given source and destination.

Booking Service: Booking Service is the microservice that lets a client place a booking request for a route at a given timeslot and date.

Cache: We have used a global cache to cache the user token for a particular user for a time window of 30 minutes.

SQL Cluster: MySQL Cluster is used for replication of database tables in order to make our service reliable.

Chapter 3: Requirements

Hosting Server: We need a host server to launch our application to make it available to the clients present in the outside network.

Caching: In order to reduce the time complexity and latency. This is being done using caching of queries involving read operations.

Reliability – We need our service to be reliable so that the clients are not affected due to any of the component failures in the system.

Database Consistency and durability - The booking application we designed required storing the user and trip details along with ensuring data consistency so that there are no conflicts and inconsistencies due to concurrent requests.

Scalability - This is one the most desired requirement of our microservices based design. Achieving this requirement will enable our services to handle increasing number of requests in a given time. Scalability is addressed on server side of the application. Our architecture is designed to optimize the consumption of server resources and grow modularly by adding more resources. Scalability of the application is implemented using dynamic approach. Number of resources addressing the query are dynamically allocated.

Availability: The architecture of Journey booking system is designed to ensure the availability at each geographical location with full time availability. Distributed system ensures redundant hardware and software that makes the system available despite of failures occurring at server’s and client’s side of architecture. As server is a distributed system, downtime of one server will be handled by diverting the requests to another available server ensuring 100% availability.

Messaging and monitoring: As we have Geo-distributed system it is really hard to monitor, measure performance and see message log of all the components at the same time. So, to overcome this problem we want to implement a feature where we can see the performance and message logs of all the components at the same place.

Chapter 4: Specification

1. Flask 1.1.2

Web Server Gateway Interface

Flask is a WSGI web-server framework. The Web Server Gateway Interface (WSGI, pronounced whiskey or WIZ-ghee) is a simple calling convention for web servers to forward requests to web applications or frameworks written in the Python programming language.

Micro-framework

Flask is a light-weight micro-framework to host a web application in python. It is known as "micro" because Flask keeps the core simple but extensible. So, Flask doesn't include "form validation" or any "database abstraction layer" or any other such features where the external extensible libraries already exist and the developer can just import those functionalities in his application as if those are implemented using Flask itself.

Explicit object instantiation

Flask explicitly creates an object instance of itself i.e., the Flask Class every time while running the application. This enables the feature to run multiple instances of the application at one time and thus enables **scaling** and **availability** of the server.

The Routing System

Flask uses the Werkzeug routing system which was designed to automatically order routes by complexity. This means that you can declare routes in arbitrary order and they will still work as expected. This is a requirement if you want to properly implement **decorator-based routing** since decorators could be fired in undefined order when the application is split into multiple modules.

Another design decision with the Werkzeug routing system is that routes in Werkzeug try to ensure that URLs are unique. Werkzeug will go quite far with that in that it will automatically redirect to a canonical URL if a route is ambiguous.

One Template Engine

Flask decides on one template engine: Jinja2. **Template Engine** evaluates a template with a set of variables and take the return value as string.

2. Redis 3.5.3

In-memory data structure store

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as dictionaries, strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyper loglogs, geospatial indexes, and streams.

3. MySQL Database

The booking application we designed has a Database server which uses MySQL database.

The user trying to book a journey should be able to do so by entering data specified in the same format as stored in the database tables below.

The user's journey data is stored in the MySQL database named "mydatabase" in the form of the following 4 tables:

- Table 1 - TRIP_DETAILS which includes the following fields: tripId ,owner_unique_Id source, destination, route_taken, vehicle_number, created_at timestamp.
  This table is used to keep track of all the bookings made for different users along with the route taken, source and destination, user's unique id and trip id.
- Table 2 - ROUTE_DETAILS with following fields: routeId, source, destination. This table separately stores the list of possible routes between source and destination in a particular geographical location.
- Table 3 -TABLE BOOKINGS_AVAIL with the following fields : timeslot, route_id, count, date_requested. This table keeps track of the bookings made till now so that the incoming bookings can be issued a journey ticket if available.
- Table 4 – USER_PROFILE with the following fields : owner_unique_Id, user_token. This table will be used for user authentication

4. MySQL Cluster 8.0

MySQL Cluster is used for replication of database tables in order to make our service reliable.

MySQL Cluster provides real-time, highly **scalable**, transactional database compliant with ACID properties- **atomicity**, **consistency**, **isolation**, and **durability**. It is built on the **N**etwork **D**ata**b**ase (NDB) storage engine. It is built around a distributed, multi-master design with no single point of failure. MySQL Cluster scales horizontally and can be accessed via both SQL and NoSQL interfaces.

A MySQL Cluster comprises of a group of computers, called "hosts", each running one or more processes, called nodes. These nodes consist of the MySQL servers in order to access to NDB data, data nodes for data storage, one or more "management servers", and other dedicated data accessing programs. The association of these components in a MySQL Cluster is depicted below:

5. Portainer:

To implement message logging and monitoring we have used Portainer which is an open-source tool for managing containerized applications.

6. Netflix Eureka:

Netflix eureka is used as a discovery service. all the microservices and the API gateway register to the eureka server using the eureka client. Once the API gateway receives the request, the gateway queries the discovery service for an instance of the application. Then the discovery service uses load-balancing algorithms and reply back the API gateway with the address of the selected instance. Now the API gateway will forward the request to this instance.

7. Spring Cloud Gateway:

Spring cloud Gateway is used to route the request to the desired service. The spring cloud gateway easily integrates with the Netflix eureka server. When they are both combined, they are used to implement client-side Load balancing.

Chapter 5: Implementation

Our journey booking service has been implemented as follows:

The available API's:

/registration: This API registers the user into our system and gives him a token which gets stored at our database against his UserId in the following format:

UserId <Primary Key>	Token
<Unique userId>	<unique UUID with prefix "token">

It is the user's responsibility to safely save his userId and token.

/route_list: This API gives a list of possible routes from a specified source and destination. The user can choose the most suitable route for him from the list and apply for a ticket for the same in our system.

```
Request params: UserId, token
```

```
Sample response:

{

        routes: ['R1', 'R2', 'R6']

}
```

/route_avail: This API returns a ticket booking number to the client. The client calls the API with the following parameters: the route he wants to take, the timeslot and the date he wants the travel pass for. The API checks the availability of the route and returns a passcode to the user if the route is available or else the user is denied a ticket if there are no slots available.

We have total 24 timeslots of 1 hour each that starts at 00.00 to 24.00 hrs and each slot is numbered serially from 1 to 24.

The maximum number of cars that each route can hold in one hour as of now is kept as constant which is not more than 50.

```
Request params: UserId, token
```

```
Sample response:

{
        passcode: "<unique 12-digit UUID string>"
}
```

How we are handling concurrent requests -

Each request to book a ticket is an atomic operation. We are maintaining a database table with the following information for the bookings –

Route ID, timeslot, date and count.

Route ID, timeslot and date makes a unique combination and count refers to the number of bookings done for this particular combination. Also, since currently we have the same maximum bound of total cars in every route so we have added a constraint to this database table on the row "count" so that the row entry never exceeds 50.

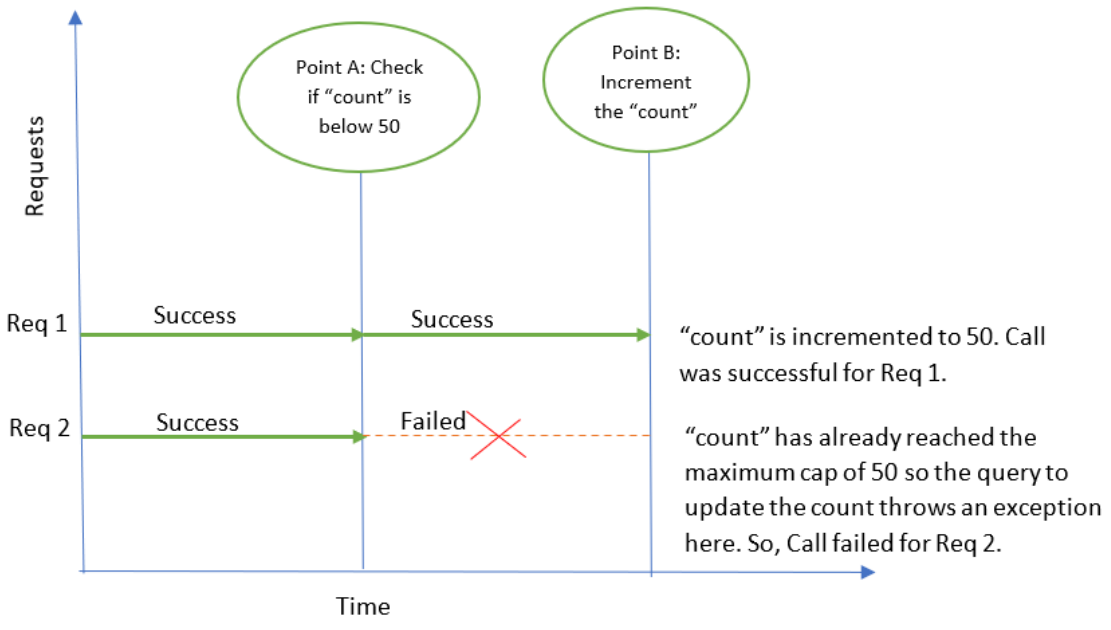
In case of concurrent requests, the query throws an exception at the point when there is a request to counter up the "count" row entry for any booking after it reached 50.

This can be show by the following example:

Current Table entry:

route	timeslot	date	count
R1	2	22-12-2020	49

Now R1 on 22nd December 2020 at timeslot 2 can take only one more request since the maximum value that "count" can take is 49


Point A: Check if "count" is below 50
Point B: Increment the "count"
Requests
Req 1
Success
Success
"count" is incremented to 50. Call was successful for Req 1.
Req 2
Success
Failed
"count" has already reached the maximum cap of 50 so the query to update the count throws an exception here. So, Call failed for Req 2.
Time

How we are ensuring user security and privacy -

User Authentication: During registration, the user gets a unique token against his userId. For every call that the user makes to our system, he is required to send us the userId and the token as request parameters. Our authentication service matches the incoming token with the token already stored at our database for the particular client and proceeds with the request if the tokens match perfectly. This is how user authentication is currently being provided.

How we are reducing time complexity -

Caching: When a user registers for our service, we provide them with a unique authentication token which he needs to pass with the API request while being logged in. We have used Redis to cache the result of the query to get the real token saved in our database for a time window of 30 minutes in order to avoid the time required to make repeated queries into our database server to authenticate the user's Identity.

How we are enhancing reliability –

MySQL Cluster and replication

To fulfil the requirements of the booking system we have used MySQL NDB cluster which is a collection of in-memory databases in a shared-nothing architecture. In our application, we have used a combination of 2 MySQL server nodes combined with 2 data storage nodes and a management server node.

The data stored in the tables created on the data storage nodes (ndb1 and ndb2) is accessible from both the MySQL nodes (namely mysql1_1 and mysql2_1) in the cluster and changes made by the application through one of the server nodes are immediately made available when queried using the other SQL server node.

Since our cluster has only 2 data nodes, each table is partitioned into 2 parts as per the default partitioning feature provided by MySQL cluster. Each data node holds a primary replica of the data stored on itself along with storing a copy of the other partition called secondary replica. Similarly, the second data node holds its own primary replica as well the secondary replica of the first data node. This means that for a cluster with 2 data nodes, each node will contain the whole database and this ensures data redundancy.

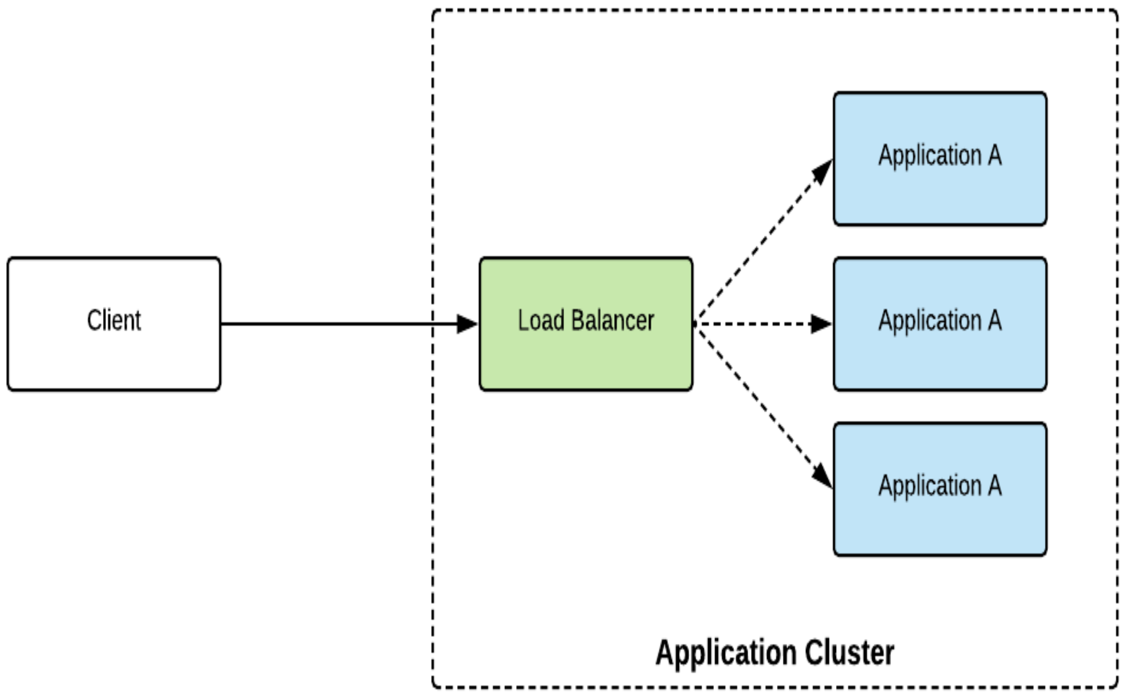
MySQL Cluster provides synchronous replication between the data nodes and this is achieved using two phase commit protocol.

Load Balancing

<u>Traditional Approach</u>

To achieve scalability and to ensure failure-safety in production, more than one instance of the application is deployed.

Now one way of reaching to these nodes is via load balancer. The client which knows the load balancer will request the load balancer. When the load balancer receives the request the load balancer will select one instance forward the request to it. The instance is selected using an algorithm most commonly round robin is used.



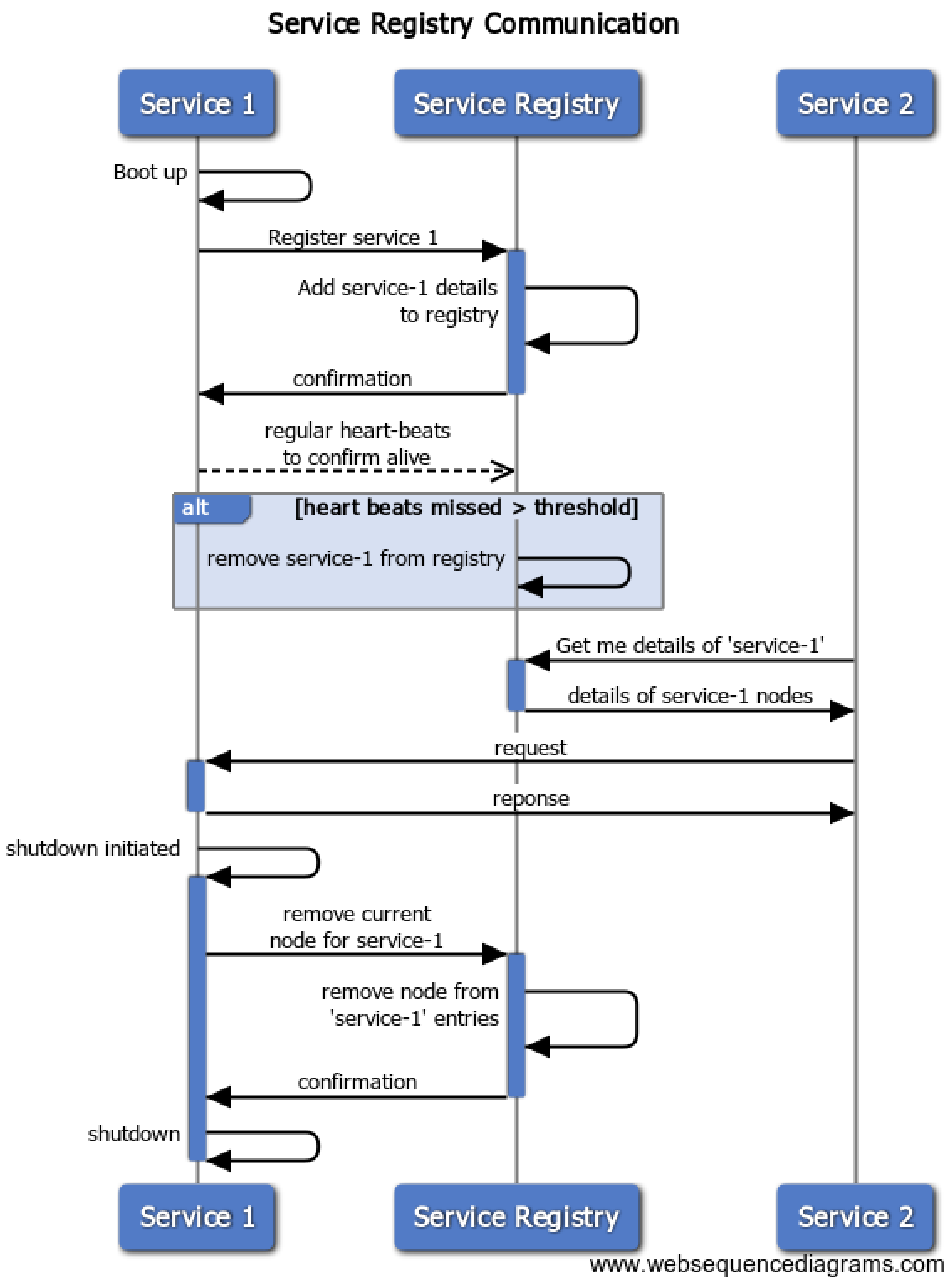
For this we need following requirements: -

1. The client should know about all the instances of the application.
2. Client has access to a component that implements normal load balancing algorithms
3. Clients just need to delegate the request to this component and it will invoke one of the instances based on some load balancing algorithm.

Client-side Load-Balancing

In this load balancing approach, the Client will handle the load balancing. The load balancing on Client Side takes care of the following aspects of our design.

1. No More Single Point of Failure in A Cluster
2. Removal of Bandwidth Bottleneck
3. Auto-discovery
4. Reduced Cost
5. Reduced Latency



Service Registry Communication
Service 1
Service Registry
Service 2
Boot up
Register service 1
Add service-1 details to registry
confirmation
regular heart-beats to confirm alive
alt
[heart beats missed > threshold]
remove service-1 from registry
Get me details of 'service-1'
details of service-1 nodes
request
reponse
shutdown initiated
remove current node for service-1
remove node from 'service-1' entries
confirmation
shutdown
Service 1
Service Registry
Service 2
www.websequencediagrams.com

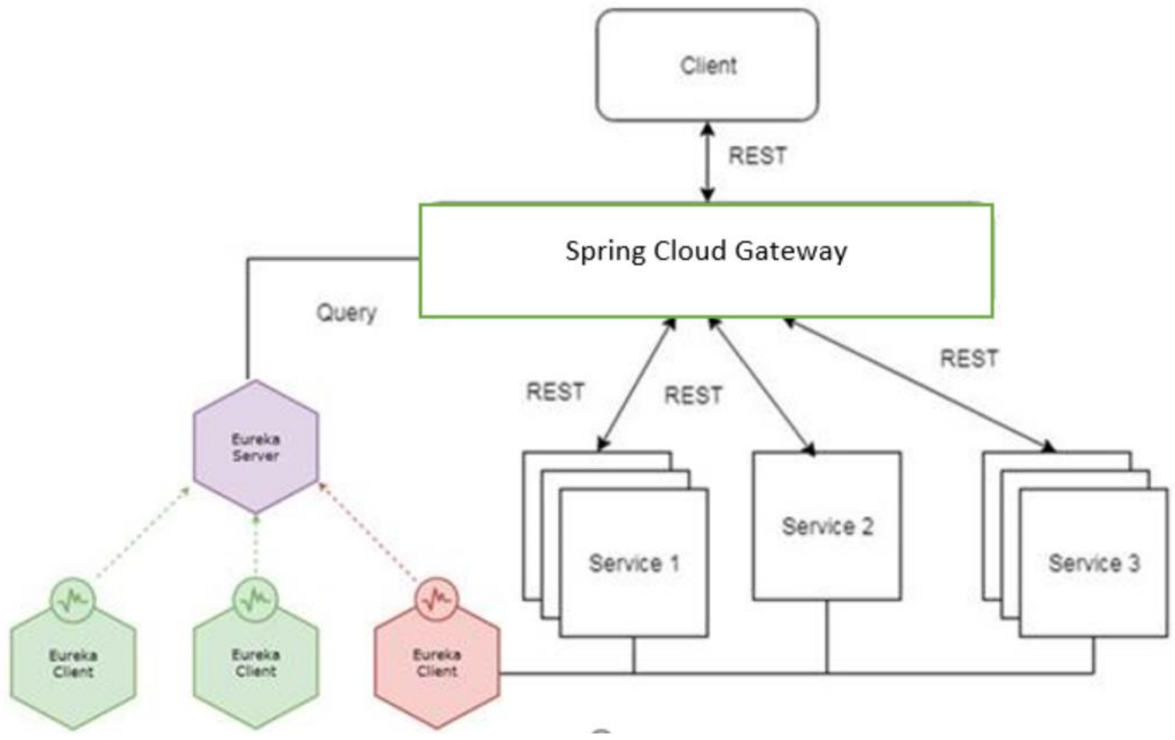
What if Service Registry Crashed?

Although we have multiple instances of Service registry but if all the instance of service registry go down at the same time, the application node still work.” This happens because the client-side load-balancer library typically keeps a snapshot of the registry data (the service IDs that it is interested in). The application nodes will use this data to invoke other services. The impact of service registry outage will be"

1. "Newly created service nodes will not be discoverable"

2. "Terminated service nodes will not be removed. However, the client-side load-balancer libraries are smart enough to detect that calls to a particular node have failed and will be eventually removed."

To use this service registry we have API gateway which is implemented using Spring cloud gateway. The gateway queries the discovery service for an instance of the application. Then the discovery service uses load balancing algorithms and reply back the API gateway with the address of the selected instance. Now the API gateway will forward the request to this instance.



Chapter 6: Testing

1. **API testing in Postman –** We are using postman as a verification tool for testing the API response for various HTTP response codes.
2. **Server Failure** – We have two MySQL servers deployed in our service. The test starts off by creating a table using one MySQL server and shutting it off. Next, while querying using the other MySQL server, the table is still accessible and hence proving no data loss due to single server failure.

3. **Conflict due to Concurrent Requests** – We have tried to update the table “bookings_avail” with two requests at the same time. The table “bookings_avail” has a column as “count” which has a constraint that the column value cannot exceed 50. The “count” specifies the number of permissible tickets for a route (Already explained in the Chapter - Implementation). Therefore, for values of count < 48, the query runs successfully for both the requests while for count = 49, the query fails for the second request and hence maintaining atomicity of the second request assuring no partial success.
4. **Service Instance failure**: If any instance of a service is down then usually there are other instances of the service which are running but since this particular instance is down then it will not send heartbeats to the eureka server and then eureka server will remove its entry and the subsequent request are forward to the live instances.

Chapter 7: Allocation of Work

The whole design and implementation of our system was equally distributed among the team members. There were regular group meetings over Teams to discuss the progress of the project on a daily basis. The work was allocated into sub groups of 2 people working together doing pair-programming.

Sub Group 1

Karan and Prabhjot were allocated implementation of the API Gateway, load balancing, dockerization of the application and messaging and monitoring GUI.

Sub Group 2

Shivani and Rupasmita were allocated the task of application workflow design and implementation using Flask framework, caching and Database design. Implementation of MySQL cluster.

Chapter 8: Summary

The whole project was full of immense learning opportunities for each of the team-members. Every team-member had a plethora of knowledge to be shared with one another and this further enhanced the collaborative learning experience. Here is a note from each of the team-members on their experiences gained from this project:

Rupasmita: This project has been full of learning opportunities for me. I have got hands-on experience in designing a distributed system ensuring the industry requirements of aspects such as scalability, reliability, availability and user privacy. I have also got the opportunity to understand containerisation and virtualisation of different microservices using docker while ensuring proper connectivity among these microservices. Another experience that I have gained from this project is the MySQL Cluster implementation and how a distributed database is actually designed at the granular level by getting involved in the whole process of integrating it with our service. Brainstorming in the design discussions with my fellow group mates to arrive at solutions to handle different failure scenarios has helped me develop and enhance my critical understanding of how to make a system fault tolerant together with the considerations I have to note while ensuring its availability in a platform with wider audience. To conclude, this project has given me a wide exposure to the aspects to be considered in designing a distributed system in order to solve a real-world problem statement.

Shivani: This project turned out to be a great learning platform and helped me apply the concepts covered in the Distributed Systems module to a real-world scenario. I was able to critically analyse the challenges a wide-scale distributed system could possibly have and how they can be mitigated. My achievements during this project would be learning about new technologies like Docker along with understanding how we can design the database to make the application robust and tolerant to data losses and failures. Going through SQL Cluster documentation and their internal architecture helped me gain insights into how real-world application databases are designed and maintained. I also learned about frameworks like Flask and how it can be used for developing a web application. This whole project helped me get an overall understanding of the various components of a distributed application and how they work together. In a nutshell, this project was a truly practical experience I achieved which I can certainly apply to succeed in my future job going forward.

Karan: In this project I learned how scalability, availability and reliability is achieved in distributed systems. I implemented a client-side load balancer using Netflix Eureka+ Spring Cloud Gateway. I learned how client-side load balancing helps in achieving availability makes our services easy to scale as new instance of the service is found by the discovery service. I also gained hands on experience in dockerizing applications and also integrating container management and monitoring tool. I tested the system in various failure scenarios and came up with the solution of maintaining availability. I also got a chance to learn more about MySQL clusters and dockerizing them.

Prabhjot: In this project I understood the major aspects of designing the architecture of a distributive system. I learnt to implement scalability, reliability, availability, replication, fault tolerance etc in the systems. In the development process I get a chance to work with Netflix Eureka+ Spring Cloud Gateway, Docker and few other techniques used in the project. Also, the development process gave me a chance to collaborate with peers and develop a system which covers majority of the failure probabilities in different parts of architecture. After testing the services and performance I was able to analyse the advantages of using a fault proof smart architecture in professional world. This project gave me a chance to perform critical analysis of the system architecture and build a set of solutions to counter visible failure scenarios that should be addressed in professional world.

Chapter 9: References

1. https://redis.io/documentation
2. https://flask.palletsprojects.com/en/1.1.x/
3. https://docs.oracle.com/cd/E19078-01/mysql/mysql-refman-5.0/mysql-cluster.html
4. https://mariadb.com/files/MySQL_Cluster_Internal_Architecture_-_MariaDB_White_Paper_-_08-26-13-001_1.pdf
5. https://www.linkedin.com/pulse/microservices-client-side-load-balancing-amit-kumar-sharma/
6. https://www.portainer.io/
7. https://github.com/mlucasdasilva/cluster-mysql/blob/master/docker-compose.yml
8. https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster-nodes-groups.html