Distributed Systems - COMP30220 Assignment 2

DDMS Project

(Distributed Delivery Management System)

Finbar Ó Deaghaidh - 18410234
Jaap de Boer - 20203421
Thomas Reilly - 18483722
Magnus Sanne - 17323853
Peter O'Donnell - 18477546
David Vasiliauskas – 20202984

Contents

1 - The Idea
Planning

When we started this project, we had many meetings discussing what the overall concept of our project would be and deciding a good concept to base our project around. Throughout the meetings we came to the following ideas.

- Review Site
- Package / Delivery system
- E-commerce site
- Social Network

We also researched which technologies would be the most interesting to explore within the given time which we also took into consideration when deciding on our idea. These are the possible technologies we researched.

- Netflix Eureka
- Netflix Zuul
- Spring boot/Cloud
- Kubernetes
- Swagger
- Java/Springboot

We decided to prioritise technologies over design idea for more exposure to new technologies and decided to implement the delivery system giving us the most scope to implement all the technologies we wanted.

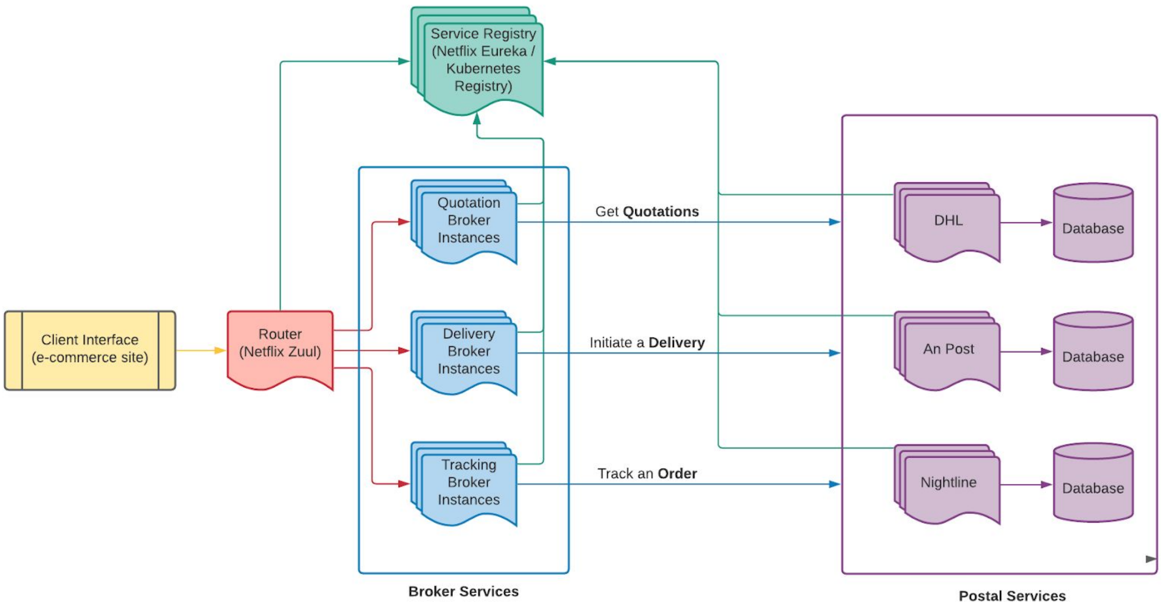
Delivery System Concept

The concept of the delivery system is that we act as a broker service between e-commerce sites and postal services. Giving E-commerce sites an API which allows them to quote, deliver and track all their parcels/orders at the most desirable price for them.

System Architecture

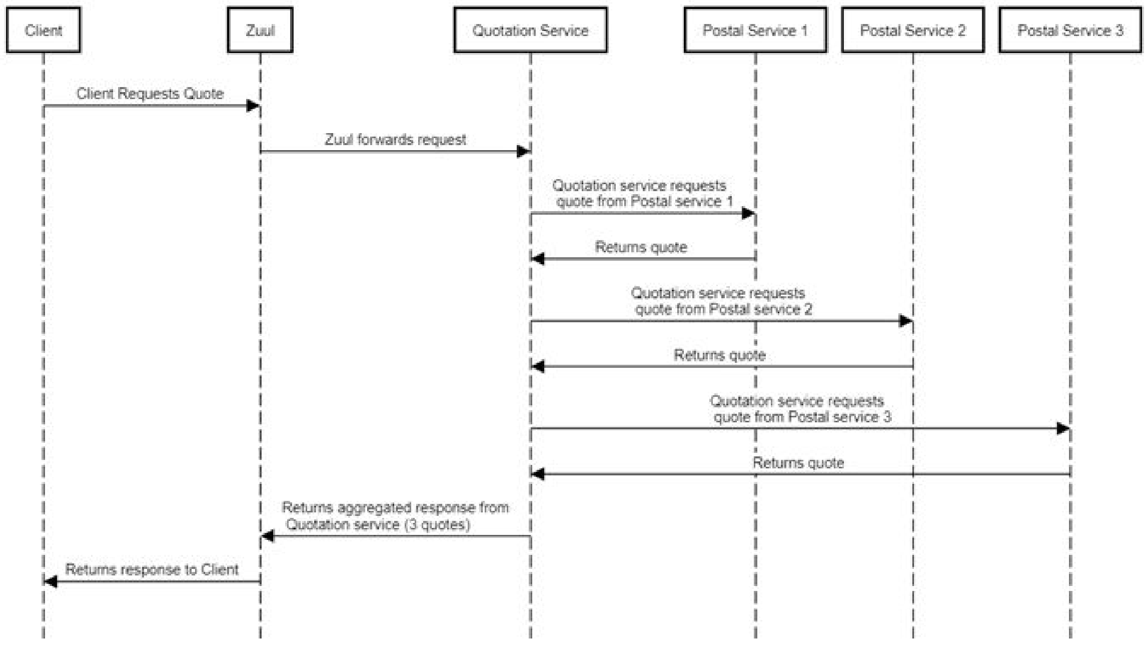
We wanted to create the project using a microservice approach, we thought the best way to implement this was to create multiple RESTful web APIs as this seems to be best practice when it comes to microservice systems.

When we were designing the system, we spent a lot of time planning what services we would need to implement to produce the working product. During the planning stage we created a system diagram and sequence diagrams to help us plan and keep track of the functionality needed to be implemented into each service.

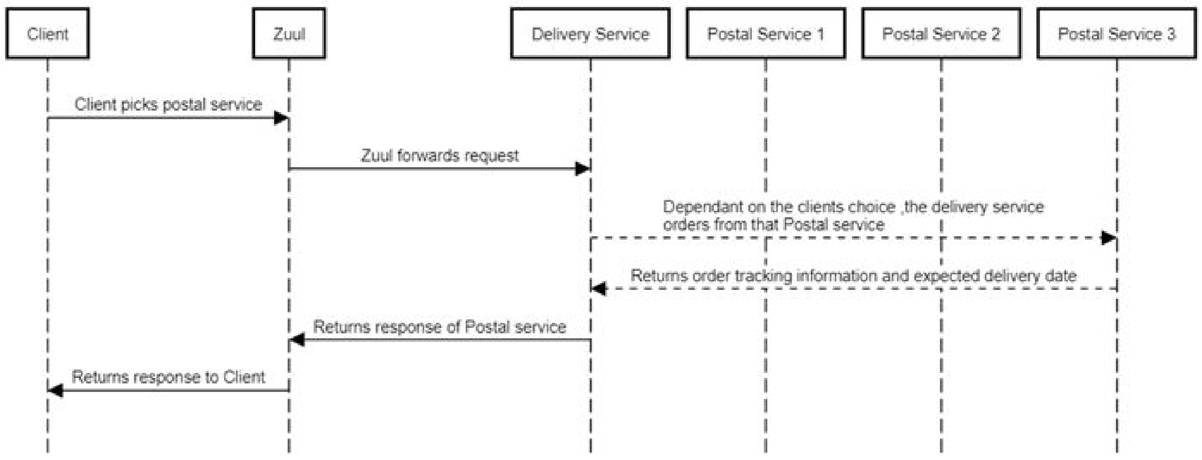

Service Registry (Netflix Eureka / Kubernetes Registry)
Quotation Broker Instances
Delivery Broker Instances
Tracking Broker Instances
Client Interface (e-commerce site)
Router (Netflix Zuul)
Get Quotations
Initiate a Delivery
Track an Order
DHL
An Post
Nightline
Database
Database
Database
Broker Services
Postal Services

This diagram shows how we decided to implement the services and the overall structure of the project. This diagram also shows how the Zuul and Eureka service will be implemented.

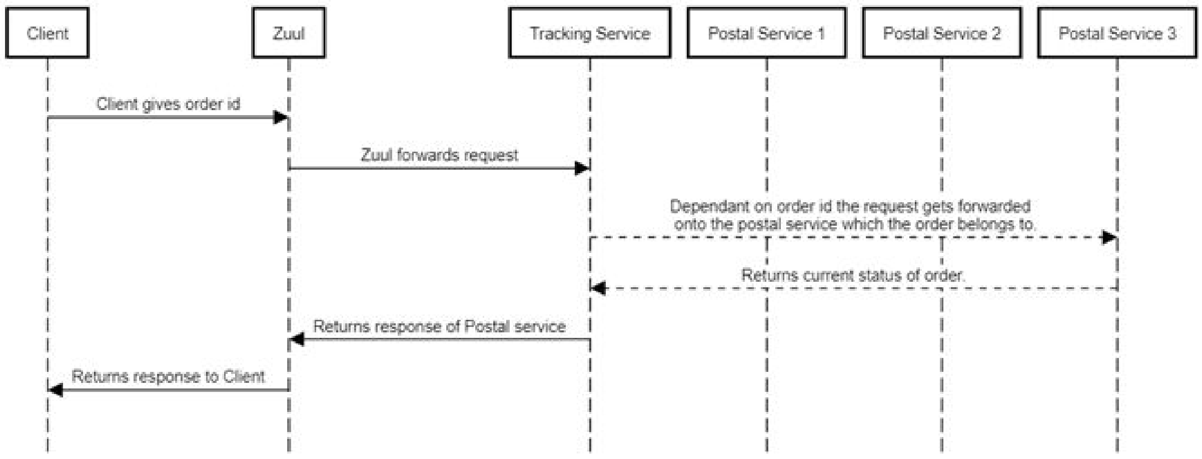
1. Quotation Requests


Client
Zuul
Quotation Service
Postal Service 1
Postal Service 2
Postal Service 3
Client Requests Quote
Zuul forwards request
Quotation service requests quote from Postal service 1
Returns quote
Quotation service requests quote from Postal service 2
Returns quote
Quotation service requests quote from Postal service 3
Returns quote
Returns aggregated response from Quotation service (3 quotes)
Returns response to Client

2. Delivery Requests


Client
Zuul
Delivery Service
Postal Service 1
Postal Service 2
Postal Service 3
Client picks postal service
Zuul forwards request
Dependant on the clients choice ,the delivery service orders from that Postal service
Returns order tracking information and expected delivery date
Returns response of Postal service
Returns response to Client

3. Tracking Requests


Client
Zuul
Tracking Service
Postal Service 1
Postal Service 2
Postal Service 3
Client gives order id
Zuul forwards request
Dependant on order id the request gets forwarded onto the postal service which the order belongs to.
Returns current status of order.
Returns response of Postal service
Returns response to Client

2 - Break Down Technology

2.a - Broker Services

Our project uses three brokers that correspond to each microservice, quotation, delivery and tracking. The brokers gather the information from the services using Json objects and send it to the client.

Quotation

The Quotation Service provides a list of quotations after being given order specifications (package details & order details). To do this the Quotation service performs a GET request to the Discovery service which returns a list of all registered postal services. The service then generates quotes by means of creating Order objects and performing a POST request to each postal service, returning a list of providers and their quotes.

Delivery

The Delivery Service provides an order date and a trackingID for an order and initiates the process of an order being shipped. It needs an order object as an argument much like the Quotation service but only needs to output the result of one postal service that has been selected via serviceID by the user (This is done from the client’s front-end in our project). The service returns the order date and tracking ID of the order specified by performing a POST request to the postal service specified.

Tracking

The Tracking Service provides the current status of an order and the facility it is at. It takes in the tracking ID acquired from the delivery service as an argument. The service returns the current location of the order, which updates as the order is completed and eventually will show that the package has been delivered. This is done by performing a GET request to the postal service.

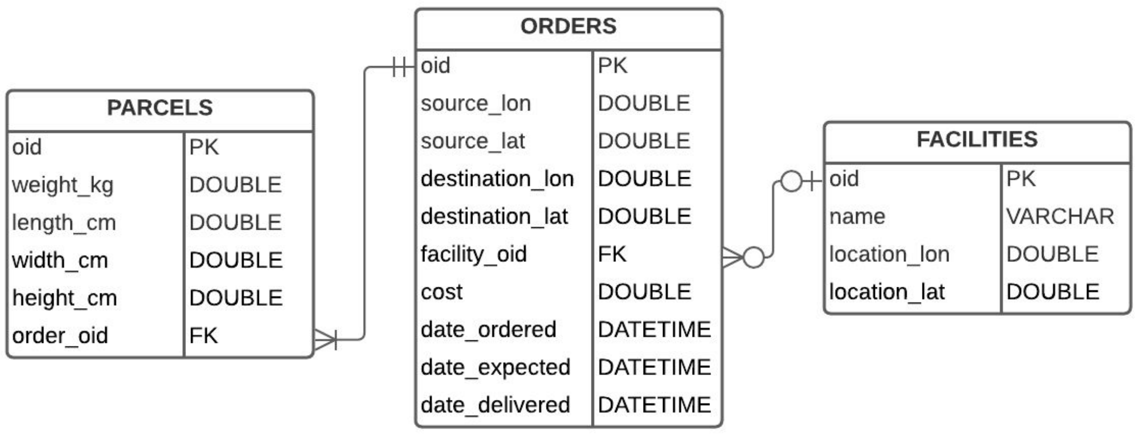
2.b - Postal Services

The project includes sample postal services which are abstractions of real life postal services like DHL and AnPost. For demonstration purposes, these services are generalizable and can be instantiated to portray differences between postal services such as company name, cost calculation and delivery speed. In real life, such companies would use their own software architecture, programming languages and database designs.

Each postal service must adhere to the endpoints listed below. Detailed descriptions of request objects and responses are also available through the OpenAPI specification and Swagger.

Route	Method	Description
/quote	POST	Request quotation for order
/orders	POST	Place a new order
/track/{trackingId}	GET	Track an existing order
/api-docs	GET	OpenAPI documentation of endpoints
/swagger.html	GET	Swagger UI of OpenAPI documentation

Postal services are scalable in the same way that the quotation, delivery and tracking brokers are. Multiple instances, or containers, of the same company can be run in parallel. Each postal service should have it's own database. Database instances of the same postal service should have their data replicated between them. If data replication becomes a bottleneck, the ratio of services to databases can be altered, but we have opted not to do this now and use a 1:1 ratio. Our implementation of the postal services use PostgreSQL databases which store all relatable information on orders such as destination, current location and parcel dimensions. See the diagram below.



As can be seen in the diagram above, our implementation of postal services uses three tables named orders, parcels and facilities. An order can contain multiple parcels and it's current location is determined by the facilities table.

The postal services also include a way of simulating packages. The simulator runs on a fixed interval and ‘moves’ packages at a predefined speed, defined in m/s. Postal companies own a set of distribution centers. Orders, or specifically its parcels, travel from source to the sorting facility closest to source. They then continue to the facility closest to the destination and finally to the destination itself. Depending on its current location, tracking requests return a status property together with its current location.

There are a number of shortcomings and improvements that could be made. We have opted to combine the tasks of generating quotes, placing orders and requesting tracking information into a single service. Further improvement, then, could be achieved with the further division of these services into sub-services, specialised in a single task mentioned above. However such improvements would be up to the individual companies to implement, but would increase the robustness and availability of such services. A shortcoming with the current implementation is that the instances of databases are not replicated. As our postal services are only for demonstration purposes this does not pose a significant problem however future implementations should take this into account to ensure persistency and consistency between instances of the same postal service.

2.c - Eureka

Netflix's open source Eureka was the central technology of our project. It exists within the discovery application and acts as a registry for all of the services, including broker services, postal services and the router service. Eureka allows for many instances of the same service to be started up and registered to it, this helps with the load balancing of requests as mentioned is section 3.d below. It also allows us to scale the number of postal-services handled by our system.

Accessing Postal Services

We needed a way for our broker services to access a list of postal services and their properties so that they knew where to forward their requests. This is achieved via endpoints in the discovery application.

Route	Method	Description
/postal-services	GET	Gives a list of all postal service instances and all of their information.
/postal-services/urls	GET	Gives a mapping of postal service names to their homepage url.
/postal-services/id	GET	Gives a mapping of postal service id's to their homepage url.

Peer Server

In the event that the discovery application fails, a replica peer service exists which also holds a Eureka server registry of all of the services. This ensures our system does not completely crash in the event that one of the discovery instances fails.

Pitfalls of our Eureka implementation

The main pitfall in this execution of service registry and discovery was that if our system came under load, instances would have to be fired up manually before load balancing could occur. Also, there was no automatic fault tolerance i.e. if a container failed it would have to be restarted manually. To remedy this we looked into kubernetes as a deployment technology (See section 3.e below).

2.d - Zuul

Netflix's open source Zuul was also used to provide dynamic routing of requests from the client as well as load balancing to our system. When multiple instances of services exist in the Eureka registry, Zuul dynamically chooses which service instance to forward its incoming request. This ensures the load caused by incoming requests is balanced among service instances.

2.e. - Kubernetes

With this technology, orchestrating containers becomes a much easier task. Provided a definition of deployments and services via yaml configuration file, scaling and management of containers can be automated.

Database considerations:
In this project, we have opted to keep Postgres database outside of kubernetes given the time constraints to deliver this project. Management of stateful instances such as a database need to be given careful attention to make sure no data loss occurs.

K8s was mostly intended to be used for stateless applications. As service instances are scaled, destroyed and re-created. Careful implementation of stateful instances must be considered.

To run databases in k8s, we would need pods to be deployed as stateful sets which get connected to persistent volumes. When a pod restarts, it reconnects to the persistent volume.

DNS service discovery:
When defining a deployment for a service with yaml files, a label should be provided. This label will be used to match a service to a request.

Internally, these labels resolve the specific pods which traffic gets routed to.

Label	**Replicas**	**Container Image**
postal-service	3	postal-service:0.0.1

Even though the ip addresses will change as postal-service gets scaled up and down, kubernetes manages this for us internally.

Difference to Zuul and Eureka:
Using K8s within this project also allows us to compare the approach and mindset of managing services with Netflix OSS: Zuul, Eureka offering vs Kubernetes.

Having load-balancing and service discovery in one platform makes it easier to manage and maintain.

Next iteration considerations:
1. Authentication between services.
2. Distributed tracing (Jaeger) with trace headers and header propagation.
3. High Availability Postgres in Kubernetes. Streaming replication with stateful instances and automatic failover.

CI & CD
In a production scenario of container and service release and development management.
On each check-in, an automated CI & CD pipeline would check the version of the service according to SemVer major, minor, patch.

Each time a developer makes changes to a service, a version gets bumped and this new version is then built, compiled and a container version of the service should then be stored in the registry. Every time Kubernetes performs a rolling release, this new version will be used from the registry.

3 - Conclusion

3.a Improvements

Overall we think the project went well but we also know we could make improvements given the time.

Databases
Each postal service only has one database container holding all information if this fails the service will fail. To fix this we could have created a DB system which replicates the data across multiple instances to improve reliability similar to how the Eureka peer works or used a service which provides this functionality like DynamoDB.

Git Workflow
Our git commits ended up very high and very messy, if we had enforced standards from the start we would have a clear git history making the repository a bit cleaner.

Eureka peer to peer
For our eureka service we currently have 2 instances of eureka running, in production it is recommended to have 3 so that if the eureka instance fails the load which is dependent on the 1st service gets distributed evenly to the two backup services while the 1st service tries to return. This stops the whole system crashing if the first service gets overloaded with requests

3.b Final Conclusion

This project was very successful and accomplishes the overall goal we set out to complete when starting this project of being exposed to new distributed technologies. We have learned two complete systems which create and run a set of distributed microservices all while working and communicating within an agile team. Distributed systems are extremely important currently in the industry and their popularity is only growing. We will all take away the skills we have gained here and most likely apply them to whatever we do in the future as developers.