School of Computer Science

A Distributed Library Management System

Report and Reflection

Team: QuocoPops

Team Members: Hassan Albujasim, Finnian Rogers, Jane Slevin

Contents

1. Synopsis
    a. Introduction
    b. Initial Thoughts
    c. Initial Architecture
2. Technology Stack
    a. Akka Actors
    b. AWS RDS
    c. Docker
3. System Overview
    a. Final Architecture Diagram
    b. Services Description
    c. Implementation Details
4. Contributions
5. Reflections
    a. Challenges we faced
    b. What we would have done differently
    c. What we learnt about the technologies used

A video overview of the application can be found at the following link:
https://drive.google.com/file/d/1hsCVWqkPsvjnu1OYBq6HGCEuthu1uTxu/view

1. Synopsis.

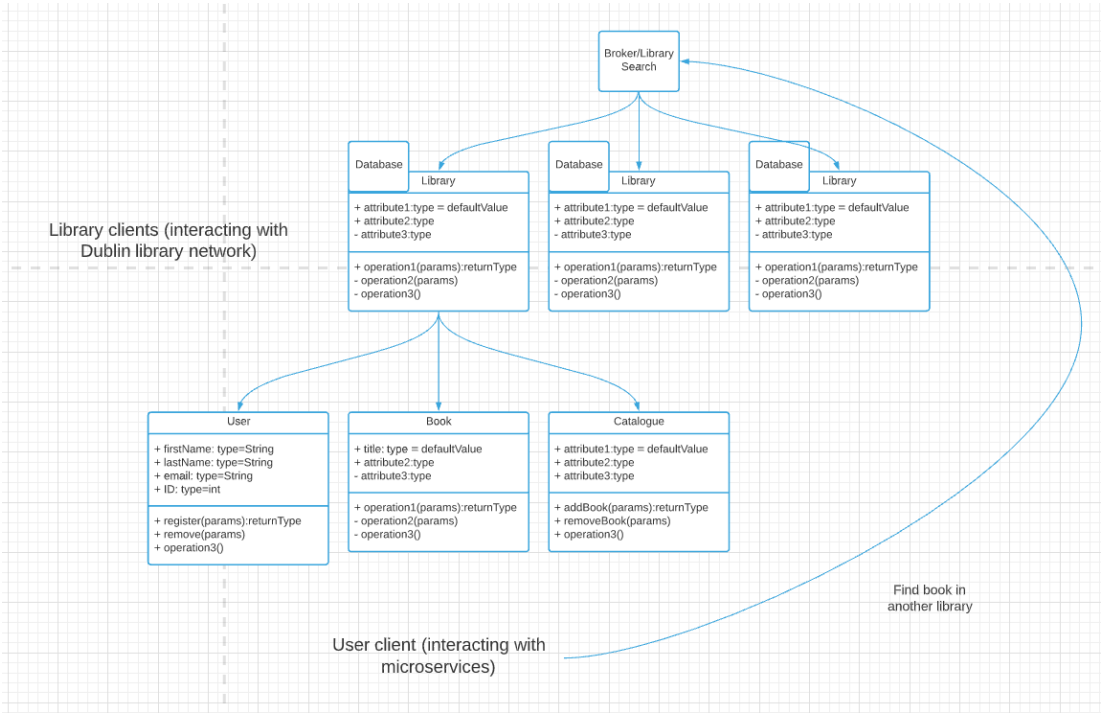
1.1 Introduction

This project was carried out by three Masters students for the Distributed Systems module at University College Dublin. The project is set to explore Actor programming with Akka, written in Java. Our aim was to explore the architecture and fault tolerance of distributed systems, by creating a backend library system that can be shared by multiple libraries. We were interested in how a system could manage more load on a particular service, how to separate the services to work independently while being able to update each other with shared resources. Finally, what might the system do if a service was shut down intentionally or by accident, how will it recover.

The three services are namely ‘Catalogue Service’, ‘Register Service’ and Loan Service’.

1.2 Initial thoughts

We initially wanted to build three separate libraries when trying to understand how the system might work. The main purpose of a shared system is to be able to search multiple libraries to find a book exists while also allowing the librarian to manage their own. Our initial sketch looked like this.



After some thought and research, we understood that libraries follow a similar system of processing in most of their services, therefore we decided to create a shared backend where libraries and their staff would be registered by reference to library name in the databases. We have set out to create independent systems, to follow closely to microservices design paradigms. *“Microservices with shared databases can’t easily scale. What is more, the*

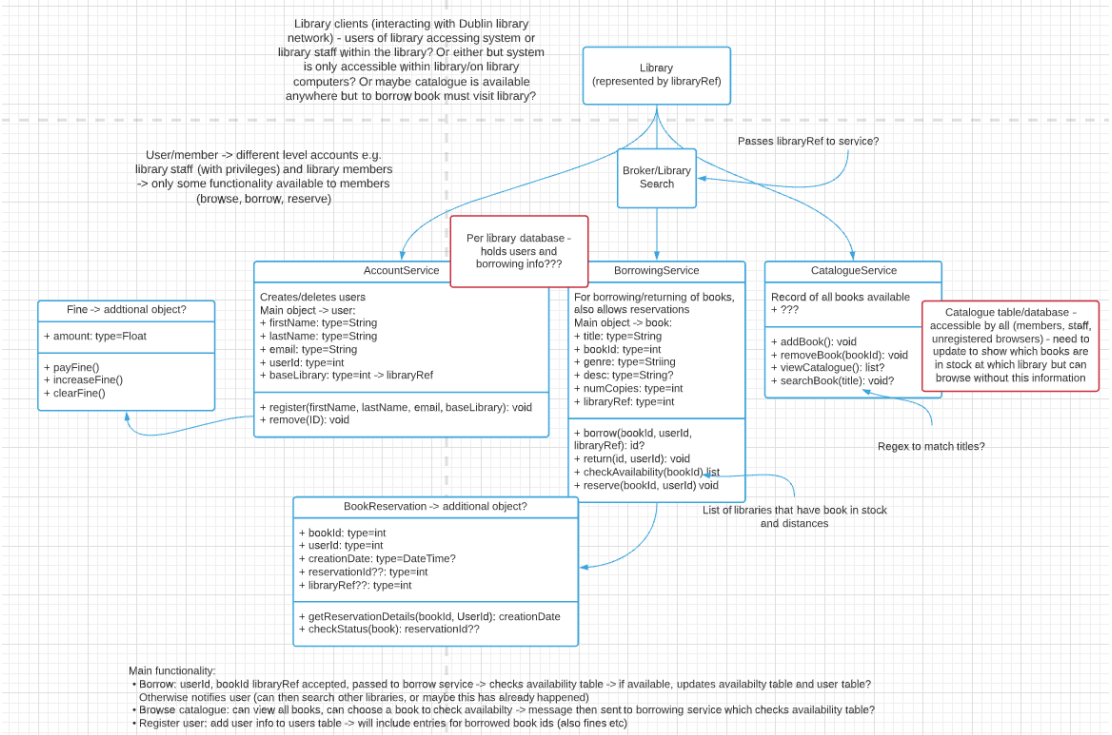
database will be a single point of failure. Changes related to the database could impact multiple services. Besides, microservices won't be independent in terms of development and deployment as they connect to and operate on the same database." Which means each system has its own database to deal with. To carry out atomic operations that should not depend on other services while mid operation. This is why we believe Actors are a suitable solution, we want the Actors to send update messages to each other periodically, of information they need to share.

Consider the following: if a user is registered in the 'Registration Service' or not how will the 'Borrowing Service' know this user_ID when scanned is a valid user of the libraries? And without having to depend on the 'Registration Service' during its check of is_user_ID.valid?

We believe that Actors would be an interesting technology to explore in this regard. I also mentioned that we are creating Microservices, the benefit of this will be to explore how to deal with load balancing of the services.

1.2 Initial Architecture Diagram

Our system would be a distributed set of microservices which would follow the below diagram. This is a sketch later down the line from the last.



We used these diagrams to discuss the pros and cons of the system we were designing; they helped in outlining the course for our work and helped us when exploring what was missing, what was the basic functionality of the system, and where we needed the services to communicate.

2. Technology Stack.

2.1 Akka Actors.

We decided to work with actors to explore this technology in more detail as it piqued our interest and felt like an intuitive approach to designing a distributed system that requires many concurrent requests. Of the technologies explored during the lab sessions for this module, we all found Akka Actor Programming relatively easy to get to grips with. This was used for the main backend logic and makes up the whole system.

Initially we wanted to implement Akka HTTP in order to add a front end to our distributed application. Unfortunately we were unsuccessful, finding the module difficult to work with (and time was not on our side). Most of the documentation and tutorials concerning Akka HTTP and any sample projects found online are written either for Scala or using Akka Typed, which is a new approach to Akka through Java. Combining Akka Typed and Akka Classic, which we were familiar with from the lab session, is possible, but we struggled to do so. Translating Akka Classic to Akka Typed also proved a difficult and time-consuming task. Our attempts to integrate Akka HTTP meant we lost a lot of time which could have been better spent elsewhere.

Having spent enough time working with Akka HTTP and with no success, we quickly decided to create a system which would carry out all of our calculations with an abstracted Client actor acting as a server. The Client coordinates the requests from different ‘clients’; these requests are hard-coded in our system and sent to the broker where they are dealt with by each related service and an output is returned to the client. In a fully functional system these requests would enter the computer from various systems in libraries and the responses would be forwarded from the client to these systems also instead of being printed out to the terminal.

2.2 AWS RDS

Scalability and fault tolerance was in our aim for the development of this project. Since we had a database connected to each service, we attempted to create docker containers with MySQL and connected one to each service container but we had “communication failure” between the application and the mysql container. With no luck in troubleshooting the issue we decided to use Amazon Web Services (AWS) cloud services to host our relational databases.

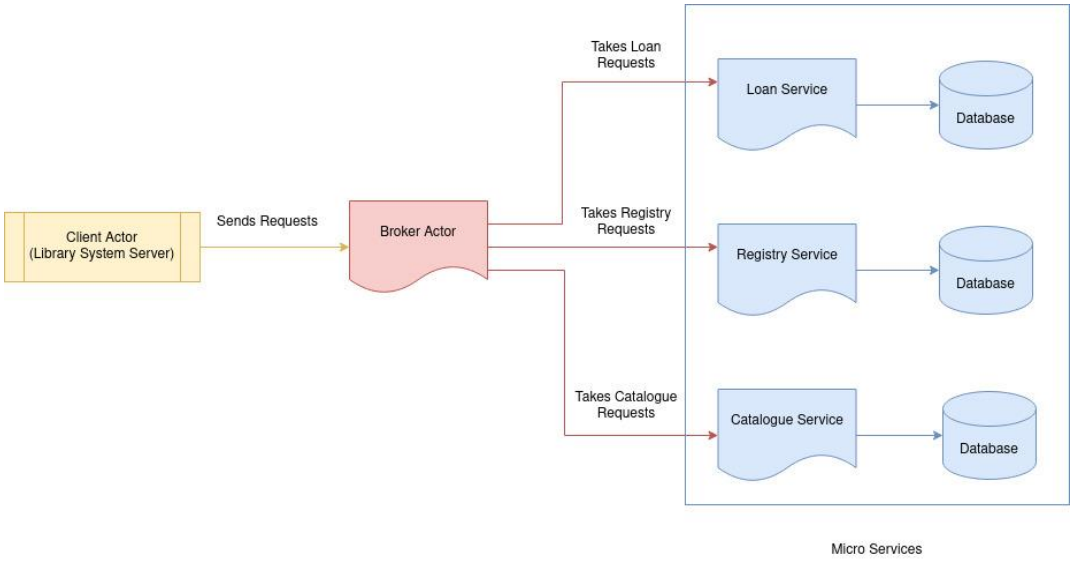
We then created stateless and scalable applications that use AWS RDS to connect to each service. Each service is connected to a private MySQL Relational Database Service instance in the cloud, which allows data persistence and which can be easily scaled as traffic increases. This would allow us to also dockerise the services for future work with Kubernetes, to allow for availability, fault tolerance and scalability.

2.3 Docker

Docker is used to create containers of each application service then with docker-compose.yml we will create the configuration and environment to run the application.

3. System Overview.

3.1 Final Architecture



3.2 Services Description

Loan Service

Simply put the 'Loan Service' will take responsibility for requests from a client to receive a book in real life. In our system we need to record the loan that took place by inserting the users_id with the book_id and returning a message with a receipt for the client. The request should contain (library_ref, user_id, book_id, timeStamp).

The broker would pass this information coming from the client to the borrowing service comi -> the service checks if the user is valid -> if yes then it insert the user_id with the book_id in a database -> compute how long the user can keep the book -> returns a response message for (library_ref, user_id, book_id, timeStamp_for_return).

Registry Service

The Registry Service is used to register library members in the system. An individual should be registered as a member in order to gain access to all functionality of the system, for example only registered members can borrow books from the library. To register, the individual provides their personal details i.e. their name, gender, year of birth, library of

registration (or home library) phone number and email address. This information is passed in a Member object (in a RegisterMemberRequest object) from the Client via the Broker to the Registry Service, which stores the data in the Registry Service database. Members can also be deleted from the database via the Registry Service, member details can be retrieved and returned to the Client and members can update their password.

Catalogue Service

The Catalogue Service is used to register/deregister new books in the system as a library obtains new books. As our system has hard-coded requests, there is no distinction between what type of user can take these actions but in a real-world system these actions should only be completable by library staff. The Catalogue Service also has a search action which takes in a book id and returns info about that book from the catalogue. Finally, the library has functionality to check book availability, which should first check if there are any copies of the book in the library you're currently in and if so, return information about the book and number of copies to the user. If the book is not available in that library but is in other libraries, then a request to the Google Maps API is made for each library containing that book to get the distance from the library the request was made from and each library that has a copy of the book. This information is returned to the user so they can see all other libraries that have the book and how far away each of them are.

Broker

The Broker collects the requested messages from the Client server actor and delivers it to the appropriate service. The Broker also shares messages between the services to add into their databases to allow for data validation. This allows for autonomy between the services and their data, which keeps inline with the microservices design paradigm.

For example the broker would send the user_id of a user to the Loan Service when they register in the Registry Service. This is so the Loan Service can have a record of the valid users and check if they can borrow an item.

Dealing with this in the broker allows for simpler database design and autonomy of computations by the services as they do not need to bother each other while they perform their requests..

Client

Our Client represents the library system server, which would collect requests from all the users and send them to the Broker. We have this abstraction as our attempts to use the Akka HTTP module failed. Our design of the Client simulates a very similar scenario of multiple concurrent requests being sent from multiple users. This is sufficient to demonstrate the system at work, but in a real-world system request messages would be taken from physical library systems to the client over a network and response messages would be forwarded from the client back to these systems instead of being printed to the terminal. Responses are returned by matching both on the library reference number and the client ID and therefore would be returned both to the correct library and the correct user within that library in a real-world system.

3.3 Implementation Details

Unit Testing

We used unit testing through our development of the system which greatly aided our development process.Through the addition of unit tests, we could easily check that an actor was behaving correctly without having the system functionality fully set up. This allowed us to work individually to build out the system component by component and eased the integration process later in the project. We used Akka TestKit to implement these unit tests. Within each test, an actor is created, a message is sent to the actor, and a test is carried out to ensure the correct response is received. In this way we could check that for each message type the actor is required to handle, it does so correctly.

Scalability

Our project is designed to scale, we have set up the services to carry their work autonomously and their database connection is private and connected to separate database instances. This allows for easier scaling of any of the services when necessary whether in load of work or database size, hypothetically for now. All services are dockerized and could be deployed in Kubernetes to create additional service nodes to scale if necessary. The databases for each service are hosted on AWS and thus are also easily scalable.

Fault Tolerance

In Akka, fault tolerance is achieved by organising the actors in a system in a hierarchy, with each parent actor responsible for the supervision of its children. If an actor encounters an error and as a result fails to handle a message, it will send a message - typically in the form of an Exception - to its parent, or supervisor, actor. When the supervisor receives such an error message, it carries out one of four possible actions: it can resume the child actor that has encountered an error, restart the child actor, stop the child actor permanently or stop itself and escalate the error to its own supervisor. The action to be taken depends on the Exception type, and is defined by the developer.

In our project we have implemented fault handling as described above within the Registry Service. To do so we made use of the Akka documentation on Classic Fault Tolerance (https://doc.akka.io/docs/akka/current/fault-tolerance.html). Within our system, a parent RegistryService actor is created when the service starts and from this four children are made, or spawned: RegisterMemberChild, DeleteMemberChild, RetrieveMemberDetailsChild and UpdatePasswordChild. As mentioned above, the Registry Service has four main functions: to register and delete members from the database, retrieve member details from the database and handle password update requests. Each child actor handles one of these functions, which removes “risky” activity i.e. connecting to and querying a database from the parent actor. The parent actor simply passes incoming requests to the correct child actors, in a similar fashion to how the broker passes requests to the correct services. On receipt of a request, each child actor will process the request and return a response to the parent RegistryService actor, which will forward the response to the Broker.

As most of the operations in our system involve database queries, the system is designed to handle SQLExceptions. If a child encounters an SQLException, it will send a message to the parent actor. If the exception is an SQLIntegrityConstraintViolationException, the parent will resume the child actor - the database query was unsuccessful but the state of the child actor is not affected so this is an appropriate response. If the exception is an SQLException, the parent will restart the child actor, reattempting a database connection. When a child is restarted, the message it was processing will be lost but all other messages in the child's mailbox will be unaffected and will be processed following restart. The behaviour for each Exception type is defined in a SupervisorStrategy object which is used within method supervisorStrategy() when an Exception is thrown. In our system, the parent will try to restart the child a maximum of ten times in one minute and if unsuccessful the child will be stopped. The values of these parameters are also set in the SupervisorStrategy object.

Tests are included in RegistryUnitTest.java which demonstrate this fault handling behaviour. The value of an Integer variable within each child actor class is updated and the value is retrieved both prior to sending an SQLException to the child and after. The variable should maintain its updated value before the Exception is sent and should assume its default value of 0 after the Exception is sent - the actor will have restarted.

4. Individual Contributions

Hassan Albujasim

My contribution was in working the high level architecture and logic of the system initially.

I contributed to writing the Loan Service code. Create aws rds instance for my service.

Created Dockerfiles for the projects and docker-compose.

I wrote the documentation. i.e I contributed to the writing of the overall report and structuring.

I created the video presentation and updated the readme presentation.

Finnian Rogers

I researched and contributed to the initial group system-architecture and tech-stack discussions/decisions.

I wrote the code for the Client and much of the code for the Catalogue Service and created and populated the AWS database for this service.

I did a lot of the initial research/coding when trying to learn Akka-Http and successfully converted the lightbend tutorial (see https://developer.lightbend.com/guides/akka-http-quickstart-java/) which is written in Akka-Typed, into Akka Classic. I was able to get the system to accept HTTP requests into the broker and forward them to the appropriate service successfully for "fire and forget" actions, but we were ultimately then unable to return a response to the client in the time we allocated for this portion of the project.

I did much of the adaptation of services and objects to ensure everything worked together once we combined our branches for each service.

I had responsibility for much of the bug fixing and ensuring everything was finalised and worked correctly in the final stages of the project.

Jane Slevin

Alongside my team members, I was initially involved in research, planning and system design.

During the development of the application I was initially responsible for creating the Registry service, which allows the registration of members in the library system. I also contributed to the Catalogue and Borrow Services, adding the CheckAvailability functions to the Catalogue Service and LoanCalculation and ReturnBookRequest functions to the Loan Service. Implementing the CheckAvailability function that checks for remote availability i.e. where the book is not available in the member's library of choice, involved working with the Google Maps API and learning how to make HTTP requests from within a Java application.

I spent some time attempting to implement Akka HTTP, as did my team members, but to no avail.

Additionally, I built the Broker to pass messages between the Client and services, with my team member who worked on the Client also contributing to the development of the Broker - the two services go hand in hand.

I handled a good deal of the communication between the services and the Broker and the Broker and the Client, in that I designed several of the request and response objects. This involved investigating what data would be required by each service to allow for data validation, e.g. only registered members can borrow a book, and to minimise communication between the microservices and communication overhead overall. The data validation that is implemented is basic and further work on this system would involve tightening control of what operations could be performed by who.

Finally, I redesigned the Registry Service, which was initially structured similarly to the other services, to implement a parent-child hierarchy allowing for fault tolerance, as described above. It was our intention to do so for each of the services but fixing existing bugs in the system took precedence in the final stages of our project. The Registry service demonstrates the potential of the system in terms of fault-tolerance.

5. Reflections.

5.1 Challenges we faced

The key challenges we faced at the beginning of our project was understanding the full structure of the project. We used diagrams and daily stand up meetings to come up with our system and reach a common understanding, where all team members were happy.

We researched distributed systems, reactive systems and consulted with the demonstrators by presenting our idea proposal. We came to the conclusion that we can create a library backend

system that could be shared by multiple libraries. It was recommended that we follow the microservices design principles which we studied and implemented to our best ability..

Another challenge we faced was dockerizing the applications with MySQL containers to create stateful applications which were not working correctly with some networking issues between the containers. We overcame this by switching the database use to AWS RDS, this made the applications stateless and we would not need disk storage for data persistence which serves to simplify the overall application architecture and also makes database scalability very simple.

5.2 What we would have done differently

Breaking up the roles earlier might have allowed for more in depth exploration. If we had one backend logic, one on dev ops to explore Kubernetes and learn implementing it and finally one to focus on the http or front end using a framework.

Overall, more ahead of time research would have helped a lot in assessing the issues that came by. The changes happening with Classic akka to akka typed made the task of using the documentation for Akka HTTP more difficult than it needed to be and this greatly slowed down our development process while trying to learn it, before eventually having to abandon it and hard-code the client due to time constraints.

More consideration of how the system would work in a real-world library before selecting our tech stack and system architecture may have also helped avoid some problems. More consideration would be necessary but perhaps an alternative approach to microservices could have been more effective for a distributed library system. The nature of how one microservice must have data about another to function is something we dealt with (e.g. having a table of valid users in the borrow service which also received messages from the broker whenever a register/deregister message was sent to the registry service), but did not seem like an ideal solution.

5.3 What we learnt about the technologies used

It felt like Akka was not very ideal for working with web applications. We did not find the http module very intuitive with our past experience and the asynchronous nature of Akka made development of a web server more awkward than we had anticipated. Perhaps a look at Play framework might have been more successful as we have used a similar system like Django and Ruby on rails. Although we expected that would be over the requirement for our project and we had already spent too much time on trying to implement akka http.

Akka seems to be going through a transitionary period from Akka-Classic which we used in the course to Akka-Typed, a new version of Akka which has rather different syntax to the Akka we were familiar with which makes it difficult to approach as beginners as the majority of the documentation for Akka-HTTP is in Akka-Typed.

Benefits of using akka programing was the easiness of how it deals with concurrent requests. Imagining that each actor is a person who deals with a particular job of the system and sends messages to others is a nice concept, exciting and easy to grasp.