COMP30220: Distributed Systems

Group Project Report

Mud Sigmoids

Evan Brierton
19374423

Nikita Skobelevs
19329563

Christian Wang
19375706

Synopsis

Our project for this assignment was building an online instant messaging application where users are able to participate in group communications in various chat rooms. The user interface of this project is a web application on which users are able to register for an account after which they may join chat rooms - or create their own - and participate by sending messages to the chats.

Technology Stack

Microservice Architecture
To distribute our application we employed a microservice architecture pattern to allow us to create loosely couples services that can run independently from one another. This is a key concept in our project to allow distribution and ensure scalability and fault-tolerance as our architecture allows us to spin up additional instances of services independently from one another during times of increased traffic. This allows us more fined grained control over our system as we can spin up extra instances of a services only for specific services that are under heavy traffic.

Docker
Docker was an obvious choice as one of the technologies to use to allow to to easily containerise our various services in an easy way to facilitate reproducible builds and environments and to allow us to quickly and easily deploy our application.

REST
The services communicate with one another using REST. This was the most obvious decision as it's simple, easy to use and can be done in essentially any programming language.

Web Sockets
We used Web Sockets to provide users with real time message delivery. This allows users that are online to receive incoming messages instantly without us having to periodically poll for new messages.

Database per Service
While not exactly a technology is it still worth mentioning that our system uses the database-per-service pattern where each microservice own its own database that only it can interact with. This is done for a couple of reasons:

1. Scalability - This allows us to scale our databases on a per service level. This is important for our project as with a messaging app you expect the number of messages to grow much faster than the number of users so this allows us to uncouple our database storing messages from our database storing user information or credentials, letting us scale them independently as traffic warrants it.
2. Fault tolerance - Having separate databases allows us to distribute them across different hosts removing the single points of failure that is present with a single monolith database.

System Overview

Our system is made up of the following services:

client
This is a Node.js server serving our React application to the user's web browser. Given this is a web application, the user's browser can be considered part of the "client" as it's making requests to other services directly from the user's machine.

api
A Java Spring Boot microservice working as the entry point for all requests coming into the system. The API service determines which service the request is for from the URL and forwards it to the correct service before returning the response back to the original caller.

auth
A Rust microservice running an Actic Web server providing a REST API to deal with user authentication. This service deals with creating user credentials when they register, verifying user credentials on login agains the database, and keeping track of session tokens.

mongo-auth
This is a MongoDB database that is used by `auth` to store user credentials.

chats
A Java Spring Boot microservice exposes an api relating to different chats (i.e. chat rooms, channels, rooms etc). This service allows for creation on new chats, stores the list of currently created chats, which users belong to any given chat and what messages have been sent to each chat.

mysql-chats
This is a MySQL database that is used by `chats` to store all the chat information.

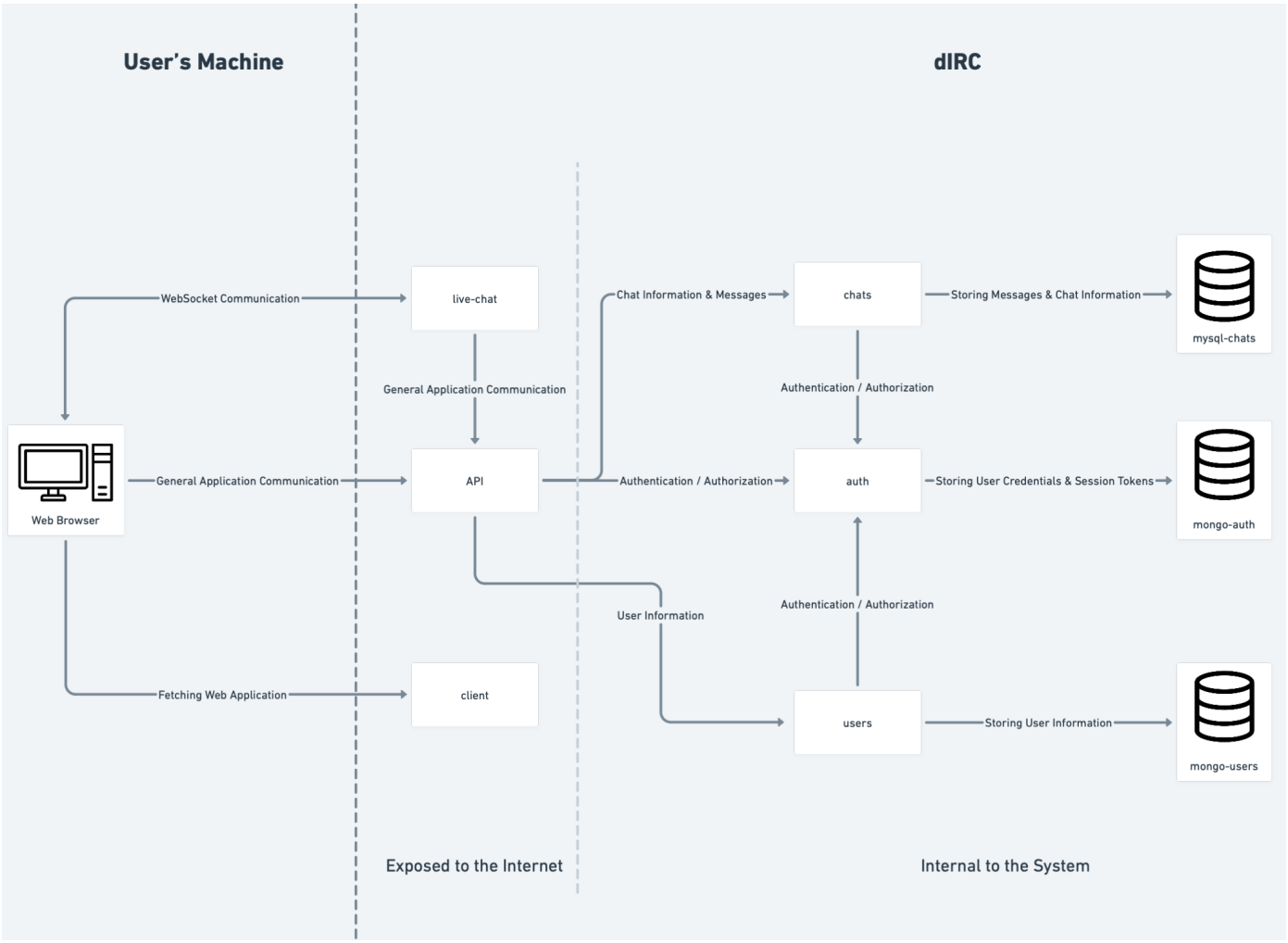
users
A rust microservice also running an Actix Web server providing a REST API to deal with all user information. Allows for creation of new users upon registration as well as fetching and modifying information of an existing user. (e.g. their username or profile picture)

mongo-user
This is a MongoDB database that is used by `users` to store user information.

live-chat
A Deno Typescript server that facilitates instant messaging by managing the WebSocket connections from all the clients. This service is responsible for broadcasting messages to all chat users whenever a message is sent, as well as notifying other services whether a message is sent or when a user leaves or joins a chat.

System Architecture

The system architecture as a whole looks as follows:

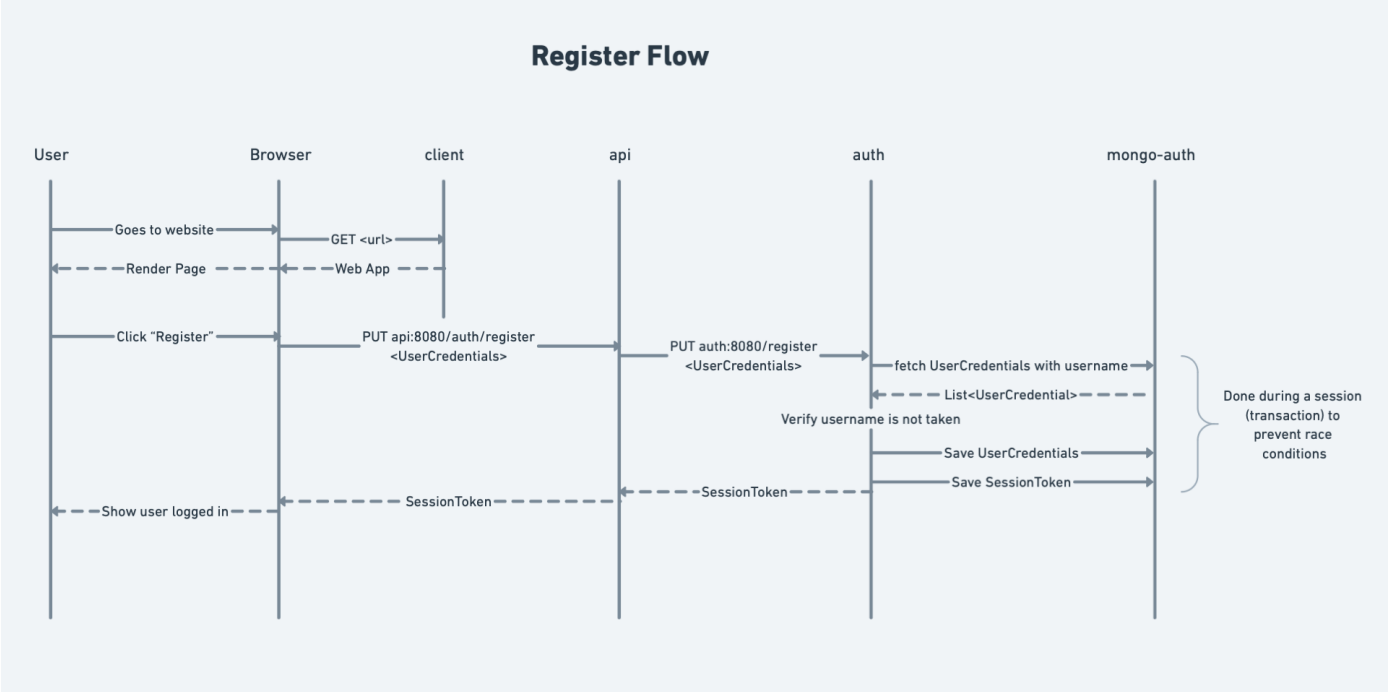


The `client`, `api` and `live-chat` services are the only ones exposed from outside the network. We have to do this for the `client` as that is what's actually serving our web content, like-wise `live-chat` is exposed to allow for Web Socket connection between it and the browser. Finally we export `api` as our "entry-point" to the system. The API takes all requests from the user application and forwards them to the appropriate service. The purpose of this service is so there is one central hostname that all requests go to without us having to explicitly specify the hostname of each service. This also has an added security benefit as we're minimising the amount of hosts directly accessible from the internet.

How the system works

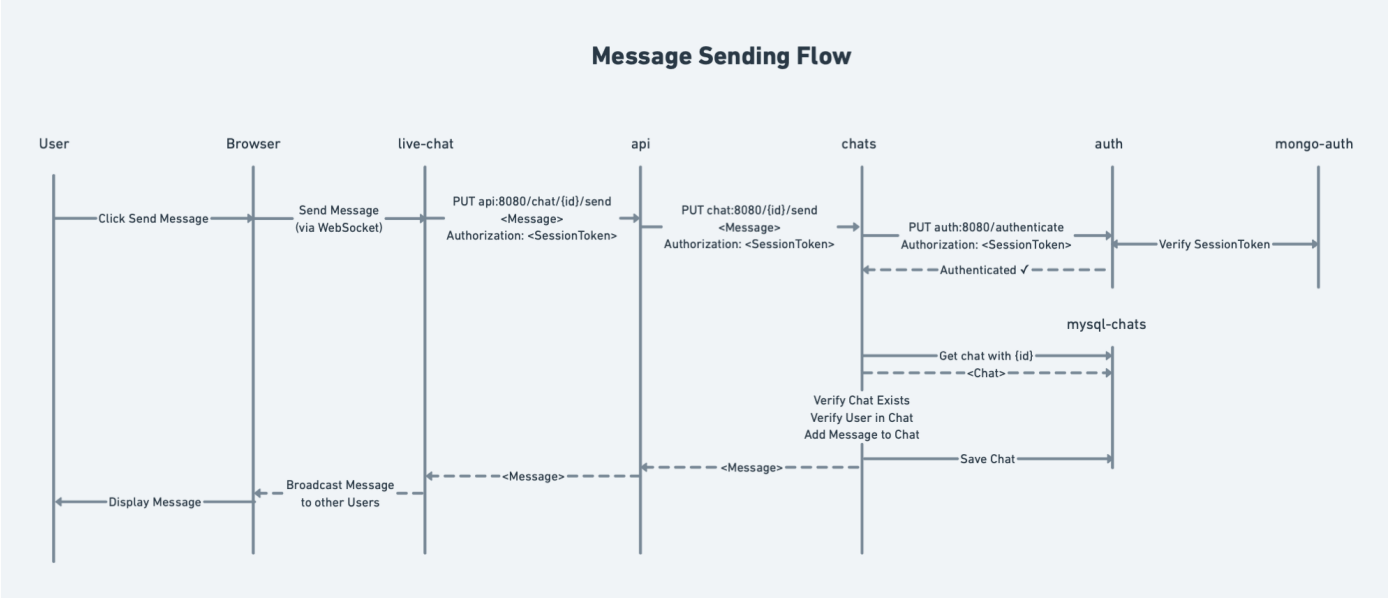
The services all work together throughout the application so there's no one clear path but we can look at two example flows to see how they all interact.

Registration Flow



We can first look at the registration flow as it's the first thing a user would do and it's one of the most simplest interactions. After the user enters their username and passwords and clicks register a request with the users details is sent to `api` that eventually reaches `auth` which makes sure the username isn't taken at which point it stores the user's login credentials (password is hashed with a random 32 byte salt) in the database. It also stores a randomly generated `SessionToken` which will be stored on the user's browser as a cookie and sent with every subsequent request to authenticate the user.

Message Sending



We can also look at the flow when a user sends a message. The request to send a message is sent over `live-chat` via WebSockets at which point it hits `api` and eventually `chats`. `chats` verifies the user is authenticated via the `auth` service. Then it checks that the chat actually exists and that the user is part of the chat after which the message is added to the chat and the updated object is stored in the database. The `Message` is returned back to `live-chat` which broadcasts it to all other users currently online with an open WebSocket connection. This ensures message delivery is instant and the front-end doesn't have to poll for new messages.

Scalability
Our microservice architecture makes this system very scalable as each service can be independently scaled with increased traffic. This scaling can be done using something like Kubernetes where each service has a corresponding Kubernetes Node which can spin up more pods of the service as needed. Kubernetes is also able to provide a load balancer for each node which allows to scale the system without us having to deal with load balancing or keeping track of instances.

As mentioned before the database-per-service also allows incredible scalability, for example after a while the number of messages stored may glow exponentially while user growth is steady. Our model allows us to scale the `chat` database independently by adding more nodes.

Fault-Tolerance
The microservice architecture also provides partial fault-tolerance where an outage of a service doesn't cause an outage of the whole system. For example the `users` service can experience an outage while having minimal impact on users - they might not be able to change their username or profile picture but the project's functionality to send messages is unhindered. Our use of Docker also makes it incredibly easy to spin up new instances of any service in case of an outage. As with scalability, the fault-tolerance can be increased with Kubernetes where each node has multiple pods running in case of an outage or crash. Similar approach can be taken with the databases - MongoDB provides built-in support for multi host database for data replication and fault-tolerance.

Contributions

Evan Brierton	
Implemented	`client`
Implemented	`live-chat`
Partial help with	`chats`

Nikita Skobelevs	
Implemented	`auth`
Implemented	`users`
Partial help with	`client`

Christian Wang	
Implemented	`api`
Implemented	`chats`

Front-end design heavily inspired by the following article:
https://blog.logrocket.com/real-time-chat-app-rust-react/

Reflections

One of the harder challenges we faces with this project was actually the logistics of figuring out what services we need and also actually implementing them independently. In our experience the interface and api of the services changed throughout the development making it a bit hard to work on something not knowing 100% what the exact requirements were. We ended up working from the bottom up, creating first the services with the least dependencies - for example `auth` was the first service built as it is independent and doesn't tightly require other services to function properly.

If we were to do this project again I think it would have been interesting to actually explore more distributed technologies. Kubernetes is the first obvious choice that would have been interesting to implement but unfortunately we weren't able to do so due to the scope of the project and other deadlines. We also briefly explored the possibility of using Apache Kafka as it's event driven messaging would be perfect for logging where each service can log to a Kafka queue and one central logging service can aggregate all the logs together. We explored this possibility and even tested it with some prototype code but unfortunately this didn't make it into our final project.

Testing Instructions

Runs with:

```
docker-compose up --build
```

Due to slower rust build times this will take several minutes to build. It will run much faster on subsequent builds due to dependency caching but the first build may take 5-10 minutes or slower for older machines.

Note: There might be an error from the users Spring service where it can't connect to mysql the first time you run docker compose. The reason for this is unknown after much debugging however if you just rerun the docker-compose everything should be fine.

Wait until all start up logging finishes before continuing.

The best way to explore the tool is using two browser windows - to have separate cookies either use two different browsers or have window in Incognito Mode.
Navigate to http://localhost:3000, you will be redirected to the register page. Registration is easy as there is no email verification, just enter a username and password (any non-empty password will do). Register for an account on each window, this will log you in.
In one window you can create a new chat by entering a chat name into the input box on the sidebar and clicking "Create Chat". The user that created the chat will be automatically added to it. You can create an invite link to the chat using the "Invite" button in the top right corner. Opening this link in the second window will prompt the user to join the chat. When both of the users have joined the chat you can effectively experience the tool - both users can send and receive messages instantly with minimal delay.