Architecture Document

Contents

[C4 Diagrams 6](#_Toc188215313)

[C1 6](#_Toc188215314)

[C2 7](#_Toc188215315)

[Impact estimates 8](#_Toc188215316)

[Technology Choices 9](#_Toc188215317)

[Spring boot 9](#_Toc188215318)

[GoLang 9](#_Toc188215319)

[gRPC 9](#_Toc188215320)

[Kafka 9](#_Toc188215321)

[React + Typescript 9](#_Toc188215322)

[SchyllaDB 10](#_Toc188215323)

[MySQL 10](#_Toc188215324)

[Kubernetes 10](#_Toc188215325)

[Cloudflare CDN 10](#_Toc188215326)

[Prometheus 11](#_Toc188215327)

[Grafana 11](#_Toc188215328)

[ELK Stack 11](#_Toc188215329)

[WebRTC 11](#_Toc188215330)

[WebSocket 11](#_Toc188215331)

[Decision Records 12](#_Toc188215332)

[Messaging Architecture 12](#_Toc188215333)

[Why 13](#_Toc188215334)

[Potential problems 14](#_Toc188215335)

[Alternatives 14](#_Toc188215336)

[Extras 15](#_Toc188215337)

[Shared Library 15](#_Toc188215338)

[Context 15](#_Toc188215339)

[Decision 15](#_Toc188215340)

[Why 15](#_Toc188215341)

[Potential Problems 16](#_Toc188215342)

[Alternatives 16](#_Toc188215343)

[Implementation 16](#_Toc188215344)

[Removal of Notification service 18](#_Toc188215345)

[Context 18](#_Toc188215346)

[Decision 19](#_Toc188215347)

[Why 19](#_Toc188215348)

[Potential Problems 19](#_Toc188215349)

[Alternatives 19](#_Toc188215350)

[Implementation 19](#_Toc188215351)

[Added Redis 19](#_Toc188215352)

[Context 19](#_Toc188215353)

[Decision 19](#_Toc188215354)

[Why 19](#_Toc188215355)

[Alternatives 19](#_Toc188215356)

[Implementation 19](#_Toc188215357)

[Add Google pub/sub 20](#_Toc188215358)

[Context 20](#_Toc188215359)

[Decision 20](#_Toc188215360)

[Why 21](#_Toc188215361)

[Alternatives 21](#_Toc188215362)

[Implementation 21](#_Toc188215363)

[Added kafka 21](#_Toc188215364)

[Context 21](#_Toc188215365)

[Decision 21](#_Toc188215366)

[Why 21](#_Toc188215367)

[Alternatives 21](#_Toc188215368)

[Implementation 22](#_Toc188215369)

[Added websocket gateway 22](#_Toc188215370)

[Context 22](#_Toc188215371)

[Decision 23](#_Toc188215372)

[Why 23](#_Toc188215373)

[Alternatives 23](#_Toc188215374)

[Implementation 23](#_Toc188215375)

[Added user presence service 24](#_Toc188215376)

[Context 24](#_Toc188215377)

[Decision 24](#_Toc188215378)

[Why 24](#_Toc188215379)

[Potential Problems 24](#_Toc188215380)

[Alternatives 24](#_Toc188215381)

[Implementation 24](#_Toc188215382)

[Permission architecture 26](#_Toc188215383)

[Context 26](#_Toc188215384)

[Decision 28](#_Toc188215385)

[Why 28](#_Toc188215386)

[Potential Problems 28](#_Toc188215387)

[Alternatives 29](#_Toc188215388)

[Implementation 29](#_Toc188215389)

[Voice Communication Architecture 32](#_Toc188215390)

[Context 32](#_Toc188215391)

[Decision 34](#_Toc188215392)

[Why 34](#_Toc188215393)

[Potential Problems 34](#_Toc188215394)

[Alternatives 34](#_Toc188215395)

[Implementation 35](#_Toc188215396)

[Adding a Turn Server 43](#_Toc188215397)

[Context 43](#_Toc188215398)

[Decision 43](#_Toc188215399)

[Why 43](#_Toc188215400)

[Potential Problems 43](#_Toc188215401)

[Alternatives 44](#_Toc188215402)

[Implementation 44](#_Toc188215403)

[Create Scylla DB cluster 47](#_Toc188215404)

[Context 47](#_Toc188215405)

[Decision 47](#_Toc188215406)

[Why 47](#_Toc188215407)

[Alternatives 47](#_Toc188215408)

[Implementation 48](#_Toc188215409)

[Added Cloud logging 50](#_Toc188215410)

[Context 50](#_Toc188215411)

[Decision 50](#_Toc188215412)

[Why 50](#_Toc188215413)

[Alternatives 50](#_Toc188215414)

[Implementation 50](#_Toc188215415)

[Added Cloud Monitoring 51](#_Toc188215416)

[Context 51](#_Toc188215417)

[Decision 51](#_Toc188215418)

[Why 51](#_Toc188215419)

[Alternatives 51](#_Toc188215420)

[Implementation 51](#_Toc188215421)

[Conclusion 51](#_Toc188215422)

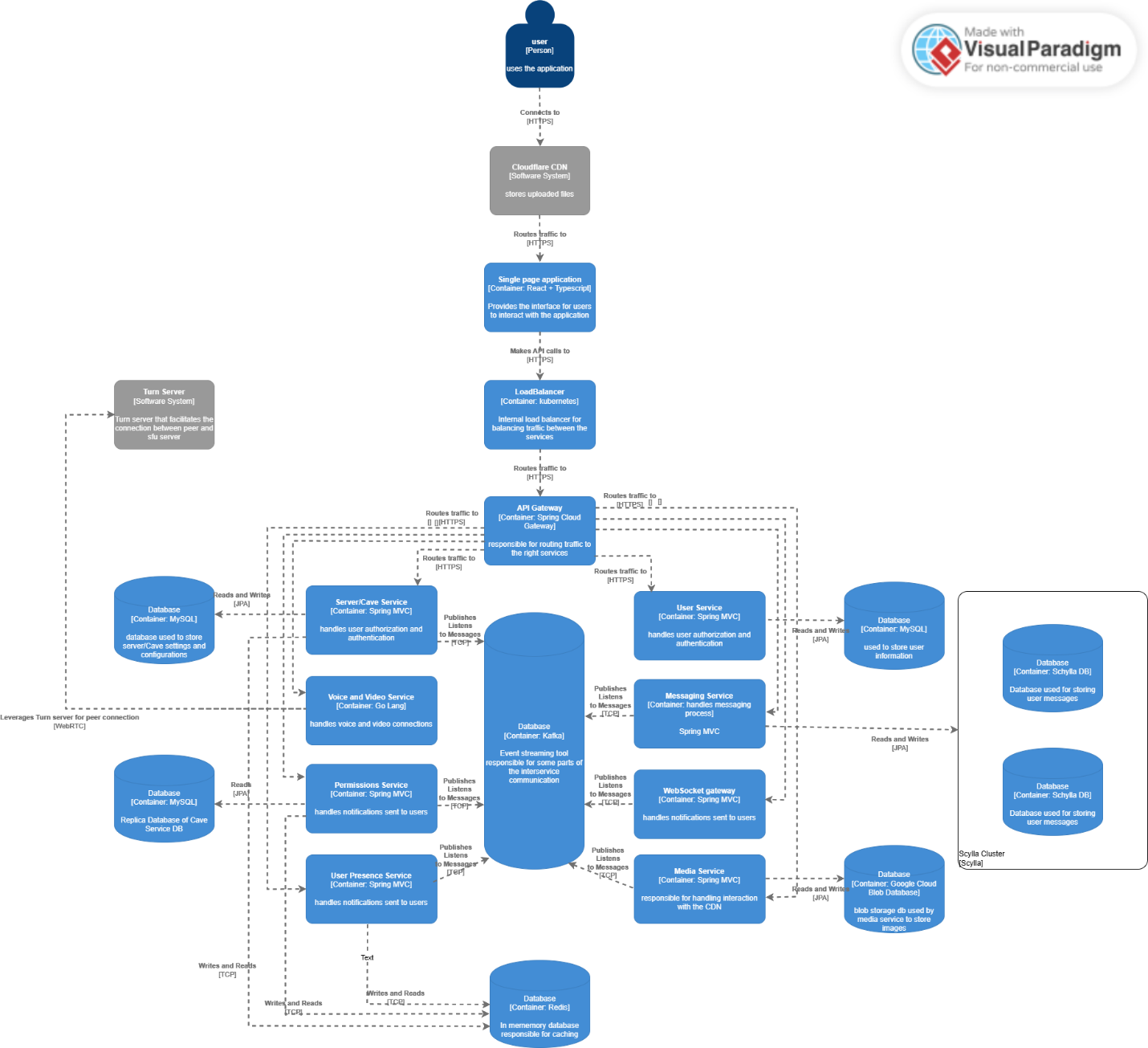
# C4 Diagrams

## C1

A diagram of a diagram

Description automatically generated

## C2



# Impact estimates

Expected concurrent user base 100K, if I this is not possible to simulate will be dropped to 50K and so on.

Connect to voice chat:

* Low latency, high bandwidth, high availability.
* 50-200KB per second per user
* Average of 8 people per voice chat
* Average of 15ms of ping from user to server

Join Server:

* Moderate latency, scalable for multiple users joining simultaneously.

Send Messages:

* Moderate latency, high throughput, scalable for high-frequency messaging.
* .5 – 1KB per message accounting message size and metadata related to it
* Hundreds / Thousands concurrent users sending messages in one single popular server.
* Average of 8 people per server sending messages simultaneously.
* The message data size majorly increases when the user sends attachments. 200KB – 2 MB

Create Server:

* Moderate latency, moderate bandwidth, moderate scalability.

Screenshare:

* High bandwidth, low latency, scalable for concurrent streams.
* Each screenshare can be between 2-4 MB per second at a HD resolution.
* Expected less traffic in comparison to voice chat and messaging

# Technology Choices

## Spring boot

Spring boot is going to be used to create most of the services due to the fast development speeds and high compatibility with most popular external systems solutions and high acceptance of microservices architecture. There is a possibility that Spring needs some adjustments on Garbage Collection because of the nature of java in high data driven application, however for an initial architecture fits the job perfectly.

## GoLang

GoLang at its core is a lower-level language however it was built to be a hybrid of both worlds meaning that has the ability of control and performance from a lower-level language but also fast development and easier syntax in comparison to other lower-level languages. GoLang’s main help for this project is the performance, it will be used to build most of the voice/video communication services providing good performance for this critical part of the application.

## gRPC

gRPC is a binary communication protocol built on top of http/2, gRPC is very good for stable, defined and fast communication between services providing sync and async communication style as well. gRPC is going to be used for communication between services where speed and consistency is of critical importance.

## Kafka

Kafka is more traditional message queue for helping provide more decoupling between the services where this is required. Kafka in this architecture will be mainly used for a more event driven approach between services that can benefit from it.

## React + Typescript

React is the most widely used frontend framework and provides all the necessary functionality I might need to develop this project. On top of this I will also use typescript instead of JavaScript because I want to keep things a little more organized and more defined instead of having the natural mess of JavaScript.

## ScyllaDB

Scylla is a C++ built NoSQL database with very high read and write performance which is critical when dealing with hundreds of thousands of operations for storing and reading messages. Also, Scylla is highly scalable and was built with high availability in mind, so it ticks all checkboxes for the requirements.

## MySQL

MySQL is a relational database with very good performance in read operations and a very good fit for structured data. Due to that it makes a good fit for more structure data portions of my services like user storage or server storage.

## Kubernetes

Kubernetes is highly standardized tool for container orchestration and a perfect solution of highly scalable systems that need high levels of management between the containers. Since the application follow’s the microservice architecture and has some services that require high levels of traffic management Kubernetes will make sure that everything is working properly.

## Cloudflare CDN

A content delivery network is a very important system when we talk about distributed file handling. The application allows users to upload files in their messages, and with this the application needs to be able to load them relatively quickly from anywhere in the world the user Is, so a CDN like Cloudflare that comes with not only the CDN part, but also highly secure traffic handling and file storage is a good fit.

## Prometheus

Prometheus will be used for metric collection throughout the services. Is a great choice because of its high performance, data throughput and horizontal scalability which is crucial in something like microservices. Also, Prometheus can be integrated with a Kubernetes cluster.

## Grafana

Grafana is a commonly used data visualization platform that is usually bundled with Prometheus. Also same as Prometheus, Grafana can be implemented in a Kubernetes cluster.

## ELK Stack

ELK stack will be used for centralized logging for all the application components.

## WebRTC

Not sure if this counts as technology choice

## WebSocket

Not sure if this counts as technology choice

# Decision Records

## Messaging Architecture

Given the use case for users being able to send and receive messages at a large scale, this is my current approach for tackling the problem.

A diagram of a software

Description automatically generated

This solution has three components:

* WebSocket Gateway:

The WebSocket’s gateway is responsible for handling any incoming events provided by the front-end application and respond accordingly.

* Message Bus:

Component responsible for handling event streaming to the required components

* Message Service:

Component responsible for processing the incoming messages and returning them if processing was successful.

When a message is sent, it's treated as an event. This is the flow it follows:

* Message to WebSocket Gateway: The message first arrives at the WebSocket gateway. The gateway then publishes the message to a message bus.
* Message Bus to Messaging Service: The messaging service, which listens for new (unprocessed) messages from the bus, picks up the message. It then processes the message and stores it in the database.
* Processed Message Back to Message Bus: Once the message is processed and stored, the messaging service publishes the processed message back to the message bus.
* WebSocket Gateway Broadcast: The WebSocket gateway also listens for processed messages from the message bus. Once it receives a processed message, the gateway transmits it to all the users connected to the channel that the message is intended for.

This way, the WebSocket gateway and the messaging service work together to handle both the processing and the delivery of messages to the correct users.

Note: The gateway knows to which user send the message because the message contains information about the channel it was sent, and the user when it selects a channel, an event is sent to the gateway to subscribe to the messages of that channel.

### Why

I do consider this solution very effective because can be scaled horizontally as much as it is needed, allowing in theory constant messaging processing times and service decoupling.

### Potential problems

Having all messages from the system running through a single messaging queue might be too much for it to handle. This an assumption that I have but I don’t know if this would be the case until I test it out.

Another problem that I don’t really like is the fact that every message is going to be listened to by every gateway instance which may be unnecessary processing, and for this I am not quite sure if there’s a better way.

AlternativesA close-up of a line

Description automatically generated

In this approach I had the WebSocket’s gateway directly connected to the messaging service. The problem that I see is that this method creates more inter service dependencies.

### Extras

Web sockets events:

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Shared Library

### Context

Throughout my application I need to check for permission from the built-in system, for that I have classes a logic that just repeat themselves in several places. This includes logic for calculating member permissions or verifying permissions against those calculated ones.

### Decision

To counter this, I have created a shared library that holds these classes and logic.

### Why

I choose to create a shared library because it allows for way less code duplication and less implementation time for this logic in new services that require it.

### Potential Problems

The only problem with having a shared library is that if it gets updated all services that use it might need to update which can lead some managing overhead, however I do consider it to still be worth it because there’s always ways to automate this update while there aren’t any straightforward ways to automate a full code implementation on a new service.

### Alternatives

Not using a library and implement the logic every time for each service.

### Implementation

For creating a shared library, we create a spring project and delete all its contents, only leaving out the main package.

A screenshot of a computer

Description automatically generated

Then we need to have a place to upload the library to, in my case I will be using GitLab package manager. Once we know where to upload, we need to configure the “build.gradle” file accordingly.

A screen shot of a computer program

Description automatically generated

Once we have that we can just run the command



A screenshot of a phone

Description automatically generated

After we publish, we can use it in any service we want. One thing to keep in mind is that this repository I’m using is private, so I need to specify on the “build.gradle” file the configurations for Gradle to import this repository.

A screen shot of a computer program

Description automatically generated

After this we can just add the library as any other and the code will be available.



## Removal of Notification service

### Context

Notification service was a service that I created in planning to have a proper notification system; however, this is no longer part of the scope, so I removed it to open time to other services that require more attention.

### Decision

Remove it.

### Why

No longer needed.

### Potential Problems

N/A

### Alternatives

N/A

### Implementation

N/A

## Added Redis

### Context

Currently there are certain parts of the application that can be deemed as heavy load operations that can happen on a regular basis but some of the times producing the same result, calculating permissions is a great example. To combat this Redis was added to provide a quick access to this type of results.

### Decision

To add Redis.

### Why

Redis industry standard caching solution that gives me all the facilities required to enhance my application. Since I’m also quite familiar with Redis it also saves development time.

### Alternatives

Memcached and dragonfly could be possible candidates to this.

### Implementation

First, I need to have a Redis instance, for this I decided to have it running on a docker container.



After, I just added the spring library for Redis and added the host and port to the “application.properties”.



A black background with white text

Description automatically generated

In my case I also setup this configuration bean for the Redis Template so that I can interact with Redis in any place I want without code duplication.

A computer screen shot of a program code

Description automatically generated

## Add Google pub/sub

### Context

I have the need to communicate certain events that happen on one service to other services where that might be relevant, for that I need some kind of event streaming solution that allows this communication while in addition being easily scalable.

### Decision

I decided not to use Google Pub/Sub.

### Why

I decided not to use Google Pub/Sub because it does not provide the functionality I require. Although Google Pub/Sub offers a highly scalable design, it lacks support for group listeners consuming messages from the same event stream, which is a critical feature I need. Given these limitations, I am considering a solution like Kafka that better aligns with my requirements.

### Alternatives

Kafka because of this group listeners approach.

RabbitMQ but lacks on the same points as google pub/sub

### Implementation

N/A

## Added kafka

### Context

The context can be found in “Add Google pub/sub”.

### Decision

Add Kafka.

### Why

I decided to add kafka for a few reasons:

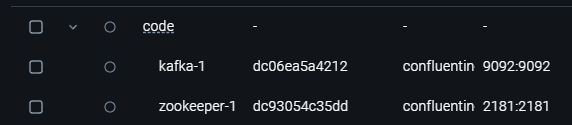
* Consumer Groups: Kafka consumer groups make it very easy to scale up individual services while keeping the consistency of functionality.
* Scalability: Kafka is a distributed event streaming solution, making it extremely scalable.
* Interest: I have high interest in trying kafka out since it’s an industry standard.

### Alternatives

The concept of kafka has alternatives like RabbitMQ or Amazon SQS, however for this project these alternatives don’t fit so well.

### Implementation

First, I need a kafka instance. For this I decided to have it running on a docker container.

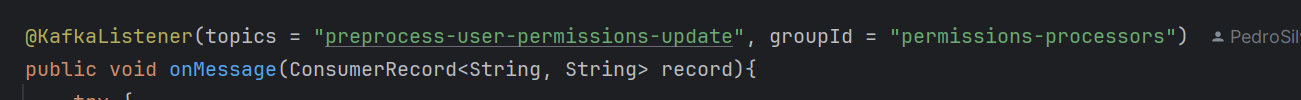


After I need to add the kafka library and specify in the “application.properties” the kafka bootstrap server.





Once we have that we can create a listener like the following.



On the other side I can also make use of the kafka template to send events.





## Added websocket gateway

### Context

The application makes a lot of use of real-time communication/events for almost all major components like messaging, updating user permissions, user presence, etc. For this there is a need to have a solution that allows for having real-time communication with the clients.

### Decision

To address this, I decided to add a websocket server.

### Why

With a websocket server I can send real-time events to any user connected to the application, this server acts as a single entry or output point for real-time events, making it behave like a gateway.

### Alternatives

GRPC could be an alternative, but it requires a substantial amount of setup in comparison to WebSocket’s.

### Implementation

In this case I’m using spring boot to create this websocket server, so I had to add the following:

A black screen with white text

Description automatically generated

A screen shot of a computer program

Description automatically generated

Now we have a running websocket server and we can connect to it.

## Added user presence service

### Context

As part of my application’s functionality is keeping track of the online and offline users, for this is required some service that is keeping track of these changes and updating the necessary values.

### Decision

I have decided to add this service due to the need of this functionality.

### Why

Currently I have no service that can handle this logic directly, so it makes sense to create a new to handle this situation, especially because this is a functionality that can have a lot of constant traffic due to possible high changes on a user’s presence.

### Potential Problems

N/A

### Alternatives

Since the user’s presence is determined by their connection to a websocket instance it could be an option handle this presence status on the websocket instance.

### Implementation

The implementation is very straightforward. Whenever a user connects to a websocket instance the instance publishes and event that will be listened by the user-presence-service. This service will update a Redis cache that contains all users and their current presence status. Once this cache is updated an event gets published so that all users that need to receive that update get it.

Websocket Gateway:

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

(Websocket listener for broadcasting the update)

User presence service:

A screen shot of a computer program

Description automatically generated

## Permission architecture

### Context

In the cave system, roles and permissions are structured hierarchically and combine both general cave-level permissions and specific channel-level permissions. This design allows for a fine-grained permission management system that is both flexible and efficient.

Roles and Permissions **Overview**

* **Cave-Level Roles and Permissions**

These are general permissions that apply to the entire cave. Examples include:

* + Viewing channels (SEE\_CHANNELS)
  + Sending messages (SEND\_MESSAGES)
  + Managing roles (MANAGE\_ROLES)

A user assigned a cave role automatically inherits the permissions defined for that role across the entire cave unless explicitly overridden.

* **Channel-Level Roles and Permissions**

These permissions are more granular and apply to specific channels within the cave.

For instance, a user may have a cave-level permission to SEND\_MESSAGES, but this can be overridden in a particular channel to disallow sending messages.

**Hierarchical Permission System**

To determine a user's effective permissions, a hierarchical approach is used:

* **Start with Cave-Level Roles**
  + Begin by aggregating all cave-level roles assigned to the user.
  + Combine these roles into a single set of permissions.
* **Override with Channel-Level Roles**
  + For each channel the user has access to, check if any channel-specific roles override the general cave-level permissions.
  + If a conflict arises (e.g., the cave role allows sending messages, but the channel role disallows it), the channel role takes precedence within that channel.

**Resulting Permissions**

The final permissions for a user are a combination of cave-level permissions and channel-specific overrides.

**Permission Calculation Using Bit Operations**

Permissions are represented as integers, where each bit in the integer corresponds to a specific permission. This approach enables fast computation and efficient storage.

* **Permission Encoding**:  
  Each permission (e.g., SEE\_CHANNELS, SEND\_MESSAGES) is assigned a unique bit position in the integer.  
  For example:
  + Bit 0 (1): SEE\_CHANNELS
  + Bit 1 (2): SEND\_MESSAGES
  + Bit 2 (4): MANAGE\_ROLES
* **Aggregating Permissions**:
  + Cave roles are combined using a bitwise OR operation (|). For example, if a user has SEE\_CHANNELS (bit 0) and SEND\_MESSAGES (bit 1), their permissions integer would be 3 (binary 11).
  + Channel-specific overrides are applied using bitwise AND and bitwise NOT to selectively enable or disable bits.
* **Benefits**:
  + Storing permissions as a single integer reduces memory usage and simplifies the process of merging roles and permissions.
  + Permission checks (e.g., "Can the user send messages?") are very fast using bitwise operations. For example:

### Decision

To tackle this situation, I decided to store the user’s calculated permissions into a Redis cache and have a service responsible for managing that. The idea is that every time a user switched their selected cave these permissions get recalculated for the selected cave and cached.

### Why

I went with this because it allows me to offload the load of constantly calculating permissions to a specified service. With Redis I additionally expect the performance to be quite good since for most of the times the permissions will be fetched from cache.

### Potential Problems

This approach adds overhead to permission checking since these calculations might be triggered quite frequently.

Having this change happening every time a user changes the selected cave might also add performance overhead because if user has cave 1 selected and goes to cave 2 the cache will update to cave 2 permissions and if he goes back to cave 1 the cache needs to be updated again, meaning that the cave 1 permissions need to be recalculated even though they most likely weren’t modified.

As always having more moving pieces adds extra room for errors.

There could be that there is a better option that I didn’t figure out in time, and since this system of permissions is quite tightly integrated with the overall services, if a better solution arises it might require time consuming refactoring.

### Alternatives

For alternatives I’m still not sure if the current solution has bottlenecks, but if the problems described above become a reality, I think there is an alternative.

First change the cache to instead of caching one set of cave permissions at a time just cache all of them and update them accordingly when required.

Secondly, I could generate the permissions for a selected cave as a jwt on a first load, this way I could just pass this token with the requests and make each service decode it and use a shared library for checking permissions logic. This would eliminate the Redis and some of the functionality of the permission service and probably make the processing faster.

### Implementation

Before showing screenshots of the code, the following classes are important to understand the structure of the cached values.

A screen shot of a computer code

Description automatically generated

A black screen with white text

Description automatically generated

When it comes to the implementation there are two main components:

* the merging of roles and permissions
* checking of permissions.

First, we have this logic for merging and fetching permissions. In this scenario I prefer to show the screenshots and use the comments as explanation because I think it is clearer.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

For checking permissions, I have the function that check a permission overall and then another function to check permissions in the context of a channel.

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

## Voice Communication Architecture

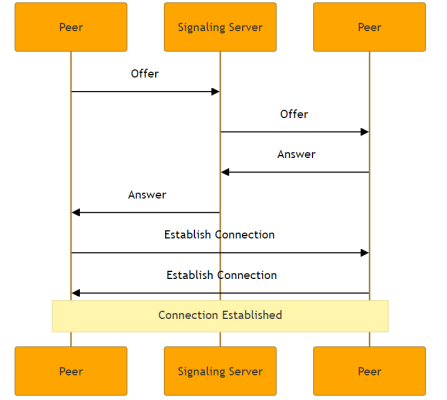
### Context

During my research on the subject, I started to see how building a scalable voice communication is a tricky subject. Discord infrastructure is based on WebRTC so that is what started with to see how it works. Very soon I realized that WebRTC is a peer-to-peer based protocol which adds a lot of trickery to this solution since by default a voice channel wouldn’t be able to pass the 10-user mark without everyone starting to have problems.

After this I started to see what I could do leverage the protocol but make the solution scalable and I came up with the following.

First there is the need to implement an SFU (selective forwarding unit) this will receive the media streams from each peer and redirect them to the peers that are subscribed to them. This removes the peer-to-peer problem since now every user from a channel can connect to an SFU and this SFU handles the data streaming. This approach also offers security opportunities since now there is no exposure of the users IP’s which is typical on a WebRTC connection.

The SFU solves the biggest problem but is not the only component. SFU handles the media redirection but there is also the need to handle the handshake for the connection.



*Example of web RTC connection*

As we can see above the SFU is technically the second peer but there is the need for this signaling process. To achieve this, I embedded this mechanism in my already existing websocket gateway ending up with something like this:

A diagram of a server

Description automatically generated

With this final piece the blueprint for my voice communication is set.

### Decision

Implement this solution.

### Why

I went with this solution because it’s very similar to what discord uses and its evident why. This solution offers very good scalability since every single component can be scaled without any problem.

### Potential Problems

Server overload. This solution is scalable in theory, but media streaming is extremely resource consuming and requires a lot of servers, so with just one of each component I wonder how far that can go without feeling server stress.

### Alternatives

An alternative would be using some kind of cloud service that offers this functionality, however that was not my goal. Also, there are multiple ways of doing this within these components, as an example I could make the signaling embedded with the SFU and remove the websocket server.

### Implementation

First let’s start with the frontend since this is what initiate the process.

The most important thing here is connectVoiceChannel function and the createPeerConnection one.

First, I get the users audio stream device. Once this stream is referenced, I call the createPeerConnection function

A screen shot of a computer program

Description automatically generated

The createPeerConnection function will create the peer configuration and add his local audio tracks to the configuration of the connection.

After this I configure two callback functions. The first is for handling ICE candidates. These ICE candidates are generated by the client and then once the first step of the handshake happens the server and the client start sending them back and forward. The ICE candidates are potential routes the peers can take to connect to each other. These routes are for example the path from local network to the NAT and so on until it reaches a clear accessible point. These routes are discovered by connecting to a [STUN server](https://getstream.io/glossary/stun-server/). (This will later change)

A screen shot of a computer program

Description automatically generated

The second callback function is handling incoming audio tracks from the SFU server. In my case I store the streams into a state and then set up an audio tag to make the audio setup.

A screen shot of a computer program

Description automatically generated

Then we go back to the connectVoiceChannel function and call the new peer configuration to create an offer. This offer contains all the information for the connection, things like information of the audio stream, audio codecs, direction attributes, etc.

A screen shot of a computer program

Description automatically generated

The last step is to send this offer to the websocket server that will send it to the SFU.

On the websocket server the offer gets here and then redirected to an SFU available.

A screen shot of a computer code

Description automatically generated

On the SFU server I receive the offer that come from the client with some data like the user Id. I use this User Id to create a new peer configuration from the server side. Then the same process that happens on the frontend happens here.

A screen shot of a computer program

Description automatically generated

The main difference is that now the SFU takes in the offer from the frontend and uses it to create an answer. This answer is the same as the offer but from the server/SFU side.

Once this is complete the connection is almost finished between client and server is complete. While this happens, other processes happen asynchronously like receiving and transmitting media tracks.

A screen shot of a computer program

Description automatically generated

The answer created by the server is now sent to the client via the Websockets server. The client receives this answer and sets it as the server answer. Now both the client and server know how to find each other and the connection is established and media tracks are exchanged

A screen shot of a computer

Description automatically generated

This is an attempt of simplifying the connection process, however this is extremely tricky to explain, and I hope it is somewhat clear. For more details I would suggest dive into the protocol and the theory behind the changes that are done to make this scalable.

On top of these there are other functionalities that are implemented:

* Channel Management
* Re-offering process
* Disconnect process

Channel Management

This is something that is implemented throughout all the SFU since any incoming connections need to be allocated to a specific group of peers (if they exist). This works by just having an HashMap that has the channel id as key and then the values are a list of the connected peers and a reference to their tracks. With this I manage separate all the peers in their virtualized channel room.

A computer screen shot of a program code

Description automatically generated

*Function that adds peer to the channel (I called them rooms)*

Re-offering process

This is the step that happens all the time whenever a user connects or disconnects. This works by sending new offers to all connected peers every time a change on a channel happens, this is required because when a user leaves a channel their audio track needs to be removed from the channel. During this process I also clean up any inconsistent peers.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A computer screen shot of text

Description automatically generated

*Signaling function*

Disconnect process

This functionality is a mix of the previous ones to since I need to remove a peer from the room and then propagate this change to all other peers by initiation the re-offer process.

Once again this is extremely complicated to explain with a lot of moving components so if you are interested, I recommend look at my research where I explain the sources I used to build this and figure it all out.

## Adding a Turn Server

### Context

As previously mentioned in my voice communication architecture one of the steps is talking to a STUN server that is used to figure out where a user is, however, this has a problem. Every single user is most likely behind a NAT, and this makes the process of figuring out the network path to reach the user tricky due to the nature of NAT, to fix this turn servers were added as an option to the protocol. A Turn server is a component that can be configured in the WebRTC connection where users send their media streams to the server and the server relays them to the other peers, in this case the peer still connects to the SFU but has its traffic relayed to the turn server and vice-versa. Turn servers usually are a failback mechanism when a direct path via STUN servers can’t be found.

### Decision

Create a turn server

### Why

For some users a turn server is required because of their NAT.

### Potential Problems

Extra overhead of having a new server and the fact that for some users this is a dependency for their communication to work.

### Alternatives

An alternative would be embedding a turn server into the SFU which removes all the overhead and combines everything into a single unit. I would have done this if I had time.

### Implementation

For this implementation I used an open-source solution called coturn.

First, we need to install coturn in a server.



*Installing coturn*

After we need to configure our server. For this there are some values we need to set.



A computer screen shot of a computer program

Description automatically generated

A screen shot of numbers and numbers

Description automatically generated

A black background with numbers and a black background

Description automatically generated

A black background with white text

Description automatically generated

A screenshot of a computer screen

Description automatically generated

Once these values are configured, we need to save the file and then start the service and make it persistent if we want.





Once this setup is finished we can try out the connection, for this I used this [website](https://webrtc.github.io/samples/src/content/peerconnection/trickle-ice/).

A screenshot of a computer

Description automatically generated

Once we confirm it works, we just need to go to the code of the SFU and the frontend and instruct it to use the turn server. I choose user turn only but turn and stun usually is the option people take.

A computer screen with text

Description automatically generated

A computer screen with many colorful text

Description automatically generated

## Create Scylla DB cluster

### Context

Until this point, I was using a single Scylla db instance inside my Kubernetes, however this is not a good practice for a production environment, at least the way I had it. To solve this, I decided to try out and implement a Scylla DB cluster with two nodes.

### Decision

I decided to implement this cluster.

### Why

The reason why I decided this is not only because of good practices, but also as a learning opportunity since database sharding is a topic I find very interesting.

### Alternatives

There are different ways of having a Scylla DB cluster. One of them is using Scylla’s cloud service which solves the problem right away with the problem of monetary costs. The other alternative would be using Scylla’s Kubernetes operator, the problem is that his is prone to a lot of error due to its complexity, but if achieved gives a almost perfect setup.

### Implementation

The first step is having the infrastructure and that means having server where Scylla can be installed. For this I have two VM in google cloud where I install Scylla. As a side note, the whole process of installation is automated in my pipeline via terraform files and Jenkins stages.

The first step setting up the Scylla repository in Linux, download and install Scylla.

A black screen with white text

Description automatically generated

After we need to configure the Scylla file. In this file there are a few values that need to be configured.

Scylla DB cluster doesn’t work like a traditional distributed MySQL, Scylla uses a peer-to-peer approach to communicate between each other. As one of the examples this makes it so that if a node goes down and another one is filled in the other nodes can setup this new node. This is a very cool system.

A black background with white text

Description automatically generated

Every node of the cluster needs to have the same cluster name configured.

A computer screen shot of a program

Description automatically generated

The second most important configuration is choosing a random IP from one of the nodes. This IP is used start that peer-to-peer setup process.

A black background with white text

Description automatically generated

*Setup the listening IP*

A screen shot of a computer

Description automatically generated

*Setting UP RPC address*

Scylla DB is an extremely powerful and efficient database for write operations, and for me in particular I really liked their approach on sharding / distributed syncing.

If I repeat this setup or two nodes, I can check then the status of the cluster.

A computer screen with numbers and letters

Description automatically generated

*Connected nodes via nodetool (provided tool by Scylla)*

Also, I have done some performance tests on this cluster that can be found on the Performance Document.

## Added Cloud logging

### Context

Originally, I had planned to build an entire infrastructure with ELK stack but once I saw google cloud offered a logging solution based on those technologies without zero setup from my side I decided to investigate it.

### Decision

Enable cloud logging

### Why

For sake of time saving and more cloud integration I decided to add it.

### Alternatives

Configuring my own logging stack.

### Implementation

For implementation the only thing required ais enable the monitoring API in google cloud.

A screenshot of a computer

Description automatically generated

## Added Cloud Monitoring

### Context

My application needed more observability and for that I made use of Google’s cloud monitoring setup.

### Decision

I decided to implement all tools and services cloud monitoring provided

### Why

The reason why I choose cloud monitoring is because I wanted to add a lot of stuff and google just facilitated that by providing all required tools and tutorials to achieve it.

### Alternatives

Cloud monitoring is made from several components like logging, tracing, profiling, metric collection and all of these have alternatives.

For example, metric collection could have been done with Prometheus mixed with Grafana for visualization.

### Implementation

The implementations for these tools and services can be found on the CI/CD document.

# Conclusion

I consider that my architecture is fairly complex with a lot of moving pieces. Even with all the other topics and time they consumed I managed to develop an architecture that I believe can be scalable to almost any requirement. I had a lot of fun and stress while building all but at the same time I learned a lot from a bunch of topics and go very deep into some of them. I would have liked to develop the voice channel capabilities even further by integrating a turn server into the SFU but unfortunately there wasn’t time. On top of that, I would also have liked to try out a more customizable observability setup with the usage of Prometheus and Grafana.

At the end I think I did a good job, and I consider that my works displays my acquired knowledge on the several subjects.