Tampere University

Alvin Wijaya, Anh Chu, Ermyas Shiferaw

PROJECT DOCUMENTATION

Student name	Student number	Email
Alvin Wijaya	152817581	alvin.wijaya@tuni.fi
Anh Chu	50358922	anh.chu@tuni.fi
Ermyas Shiferaw	150392820	ermyas.shiferaw@tuni.fi

Group name: Advance Backend
Gitlab URL: https://course-gitlab.tuni.fi/compcs510-spring2025/advance-backend

April 2025

CONTENTS

System Documentation

Architecture

The system follows a microservices-based event-driven architecture using Kafka as the central messaging system.

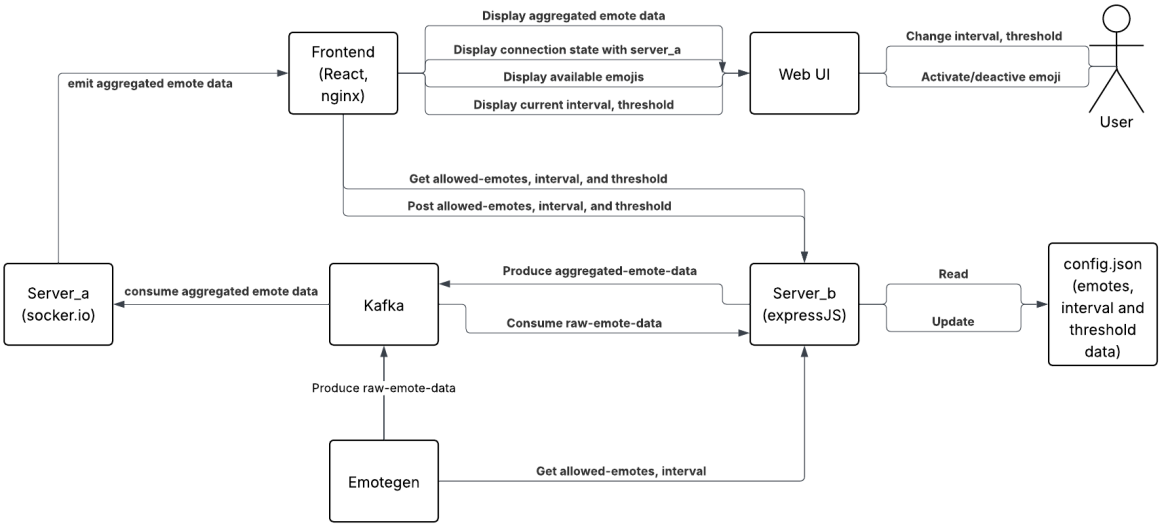

Image 1. Project's architectural design UML

The architecture consists of the following components:

Components

1. **Server A (WebSocket Server)**
    - Listens on the Kafka topic `aggregated-emote-data`.
    - Forward received messages to connected frontend clients via WebSockets.
2. **Server B (Analysis & REST API Server)**
    - Consumes data from the Kafka topic `raw-emote-data`.
    - Processes the data to determine significant moments.
    - Publishes results to the Kafka topic `aggregated-emote-data`.
    - Exposes a REST API to get and set processing thresholds, intervals, and available emotes.

3. **Kafka (Message Broker)**
    - Handles communication between services asynchronously.
    - Topics Used:
        - `raw-emote-data`: stores raw input events.
        - `aggregated-emote-data`: stores processed events for frontend consumption.
4. **Frontend (Client Application)**
    - Communicates with Server A via WebSockets to be updated in real-time.
    - Provides user interface to visualize.
    - Sends configuration update of Server B via REST API.

Applied Patterns

The project's applied design patterns is defined when the project first initialized and later while the project is progressing. The applied design patterns based on the technologies such as docker and Kafka which produces the tools such as containers and message broker tool.

- **Event-Driven Architecture**: Splits components using Kafka for message passing.
- **Microservices Pattern**: Each service (Server A, Server B) runs independently.
- **WebSockets for Real-time Updates**: Ensures low-latency data transfer to the frontend.
- **REST API for Configuration**: Provides a way to dynamically alter system behavior.
- **Pubslish/Subscribe pattern**: Kafka provides message broker which services can publish data to topics and subscribe to topics.

Technologies Used

For this project, a small variety of libraries are selected to optimize the space in the containers.

- **Backend**
    - **Node.js** with Express.js for REST API (Server B)
    - **Socket.io server (WebSocket library for Node.js)** for Server A
    - **Kafka (Apache Kafka)** for message brokering

- **Frontend**
  - **JavaScript (React)** for the user interface
  - **Socket.io client** for real-time data updates
- **Both Frontend and Backend**
  - **Docker** for creating individual image and container for each service
  - **Docker compose** for combining all individual containers and creating network between them

How to Try the System

Prerequisites

Before installing the system, these requirements need to be in the local machine:

- Node.js installed
- Docker (for Kafka setup)
- A Kafka broker running with the required topics created

Steps to Run

Run the entire system:

```
docker compose build && docker compose up -d
```

Run and test Server B (REST API & Analysis Server):

```
docker compose build && docker compose up -d
```

Go to browser and go to: http://localhost:8000

Run and test Server A (WebSocket Server):

The server A will need socker.io-client package to listen to the event.

1. Install test dependency (run this first)

```
npm install socket.io-client
```

2. Create and populate the test file server_a/test.js

```
const { io } = require("socket.io-client");

const URL = "http://localhost:3001"

const socket = io(URL)

socket.on("connect", (data) => {
    console.log("Client connected")
});

socket.on("rawEmoteData", (data) => {
    console.log(`Raw emote data: ${data.value}`)
})
```

3. Build and start services

```
docker compose build && docker compose up -d
```

4. Run the test (a new terminal for each command after services are up)

This command is to test the `aggregated-emote-data`

```
docker exec -it advance-backend-kafka-1 kafka-topics.sh --bootstrap-server
localhost:9092 --list results: raw-emote-data aggregated-emote-data
```

This command is to test the `raw-emote-data`

```
docker exec -it advance-backend-kafka-1 kafka-console-consumer.sh --
bootstrap-server localhost:9092 --topic raw-emote-data --from-beginning
```

Run and test the Emotegen:

1. Build and start services

```
docker compose build && docker compose up -d
```

2. Run the test (in a new terminal or after services are up)

```
node server_a/test.js
```

Go to browser and go to: http://localhost:3000

Run and test the Frontend:

```
docker compose build && docker compose up
```

Go to browser and go to: http://localhost:3000

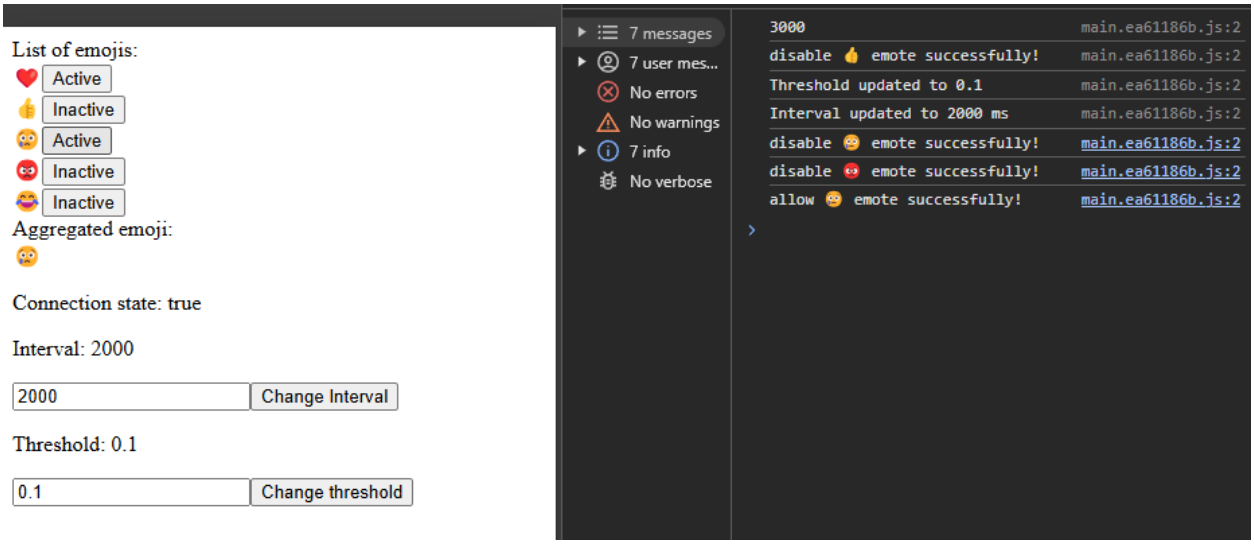

Image 1. Frontend Display with Browser Console

After opening the frontend page by going to http://localhost:3000, you will see a real-time interface to use with the emote system. Near the top of the page, you will see a section displaying the list of emojis (emotes) that are currently allowed. A button is present next to every emoji that indicates whether the emoji is "Active" or "Inactive." When this button is clicked, its status will be toggled and an update will be sent to the backend immediately through the postEmotes API. This allows you to enable or disable emojis in real time.

Underneath that, there is a section called "Aggregated emoji," which displays the stream of emojis being processed and received by the frontend through a WebSocket connection. These emojis are reactions aggregated and gathered from user input and pushed to the frontend from the server.

Below these are two input panels. The first one is for updating the interval, which specifies how often the frontend receives or processes emote data. It shows the current interval and contains an input box where you can put in a new value (in seconds). Click the "Change Interval" button after inputting the new value to effect the change and submit it to the backend through the postInterval API.

The second panel is to alter the threshold, which accepts a float between 0 and 1. This likely defines the lowest frequency or weight to display grouped emojis. Like the interval section, it has

an input box to enter a new threshold and a "Change threshold" button that makes the change through the postThreshold API.

At the bottom of the interface, the Connection State is displayed. This indicates whether the frontend is currently connected to the WebSocket server. It will display "true" if connected and "false" otherwise, allowing users to more easily identify connection issues with the real-time backend.

The system should now be live, displaying real-time processed events at the frontend as well as being configurable via the API.

Timetable

The initial timetable has 3 major steps: research, design, implementation

In the research stage, the research topic will mainly about new technologies and known technologies, but we need to understand more clearly and have more practices.

In the design state, the architectural design is decided in the meeting and drawn in the UML chart.

In the implementation stage, all the services were implemented based on architectural design.

Step	Weeks	Tasks	Responsible person
Research	1 week (17.3 – 21.3)	Research on what is Kafka, and what is it used for	Alvin, Anh, Ermyas
		Research on what the emote generator do	Alvin
		Research on what the server_a do	Anh
		Research on what WebSocket is and how WebSocket works	Anh
		Research on what the server_b do	Ermyas
		Research on what REST API is and how does it works compared to WebSocket	Ermyas

		Research on what frontend's framework will be suitable (Chosen one is React)	Anh
		Research how docker and docker compose work	Alvin, Anh, Ermyas
Design	1 week (24.3 – 28.3)	Decide what architecture will be used in the project	Alvin, Anh, Ermyas
		Design the architecture in the UML	Anh
		Design the frontend UI	Alvin
		Create the Gitlab issue boards and create Gitlab issues	Alvin
Implementation	4 weeks (31.3 – 25.4)	Emote generator implementation	Alvin
		Docker compose initialization	Alvin
		Kafka implementation	Alvin
		Server_a implementation with web socket	Anh
		Frontend implementation with React	Anh
		Server_b implementation with expressJS	Ermyas
		Create API gateway for frontend to fetch and post data	Ermyas
		Analyze and send aggregated emotes via kafka	Ermyas
		Fetch API from server_b to frontend and listen to aggregated emotes data	Anh
		Write error handling for services	Ermyas, Anh
		Connect server_b with emote generator	Ermyas
		Write documentation	Alvin, Ermyas, Anh

Table 1: Timetable for the project

Every group member promises to work on the project at least 5-7 hours a week and we kept track of the project's status in the GitLab's issue boards.

Gitlab issue board

In our issue board, we divided the column of the issue board into 4 stages: Open, progress, review, and closed

In the open stage, most of the issues are initially listed there and few more issues were created later while the project is progressing

In the progress stage, the issues will be taken from the open stage to start the issue and implementation. A branch will be created for each issue.

In the review stage, the branch will be made with a pull request, then other team's members will check the branch and make the merge to the main branch.

And finally, when the branch is merged to the main branch, then the issue will be closed.

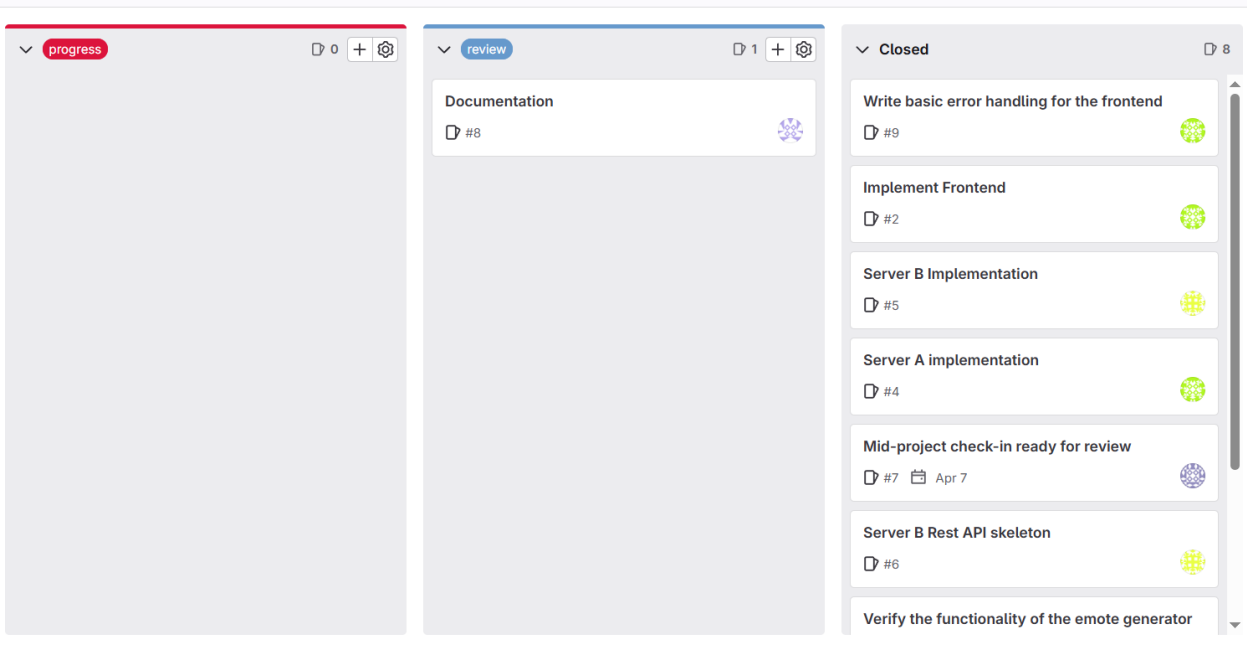

Image 2: Almost finished Gitlab issue board

Lesson Learned

We have learned how to use new tools such as Docker, Docker compose and Kafka. These tools provide a unique way of developing software product which make the product

scalable and maintainable. We have learned how to download node image and Kafka image using docker. We investigate how to create network dependencies between each service so that they can run sequentially and communicate with each other well.

We have learned to use Kafka as a message broker which uses publish/subscribe design pattern. We have learned how to implement Web socket and REST API. We can definitely compare how they different from each other and what are the strengths and weaknesses of each method. We have learned backend's and frontend's framework such as Nginx, React, ExpressJS.

UML design and selecting suitable architecture is also an important lesson. Based on the architecture, the services are easily divided, and each member will focus on his own specific service. How to write clean and optimized code with some basic error handling is always preferable.

Additional Feature

We have included basic error handling for both the backend and frontend. Also, instead display the data as Json file in the web UI, we have displayed it with emotes and also include some buttons to activate and deactivate those emotes.