Emote System Documentation

Architecture Overview

This document describes the architecture and implementation of the Emote System, a distributed application that enables viewers of live streams to participate by reacting to meaningful moments and allows broadcasters to analyze viewer reactions.

Components

The system consists of the following components:

1. **Emote Generator**: Produces random emote data with timestamps
2. **Server B**: Analyzes raw emote data to identify significant moments
3. **Server A**: Delivers significant moments to the frontend via WebSocket
4. **Frontend**: Displays significant moments and provides settings management
5. **Kafka**: Message broker for communication between components

Technologies Used

- **Backend**: Node.js with Express
- **Frontend**: React
- **Message Broker**: Kafka
- **Containerization**: Docker and Docker Compose
- **Web Server**: Nginx (for frontend)

Communication Flow

1. Emote Generator produces random emote data and sends it to the `raw-emote-data` Kafka topic
2. Server B consumes the raw emote data, analyzes it to identify significant moments, and sends them to the `aggregated-emote-data` Kafka topic
3. Server A consumes the aggregated emote data and sends it to the frontend via WebSocket
4. Frontend displays the significant moments and allows users to manage settings via REST API calls to Server B

How to Run the System

Prerequisites

- Docker and Docker Compose installed

Steps to Run

1. Clone the repository
2. Navigate to the project directory
3. Run the following command to start all services:

`bash docker-compose up -d`

1. Access the frontend at http://localhost:8080

Stopping the System

To stop all services, run:

`bash docker-compose down`

Component Details

Emote Generator

The Emote Generator produces random emote data with the following characteristics: - Generates a new emote every second - 80% chance for a single emote - 20% chance for a burst of multiple (same) emotes - Each emote includes a timestamp

Server B

Server B is responsible for: - Consuming raw emote data from Kafka - Analyzing the data to identify significant moments - Sending significant moments to Kafka - Providing a REST API for settings management

REST API Endpoints

- `GET /settings/interval` : Get the current interval setting
- `PUT /settings/interval` : Update the interval setting
- `GET /settings/threshold` : Get the current threshold setting
- `PUT /settings/threshold` : Update the threshold setting
- `GET /settings/allowed-emotes` : Get the current allowed emotes
- `PUT /settings/allowed-emotes` : Update the allowed emotes

Server A

Server A is responsible for: - Consuming aggregated emote data from Kafka - Sending the data to the frontend via WebSocket

Frontend

The frontend provides: - Real-time display of significant moments - Settings management interface - Animations for significant moments

Design Patterns

Publish-Subscribe Pattern

The system uses the publish-subscribe pattern through Kafka for asynchronous communication between components. This allows for loose coupling and scalability.

MVC Pattern

The frontend follows the Model-View-Controller pattern with React components for the view, state management for the model, and event handlers for the controller.

RESTful API

Server B provides a RESTful API for settings management, following REST principles for resource manipulation.

Deployment

The system is containerized using Docker and orchestrated with Docker Compose. Each component runs in its own container, and they communicate through a shared Docker network.

Future Improvements

- Add authentication for the settings API
- Implement persistent storage for settings and significant moments
- Add more sophisticated analysis algorithms for identifying significant moments
- Enhance the frontend with more interactive features