Design Documentation
Event On Group Chat

Introduction

Purpose:

The purpose of this software is to imitate human conversation between members of a group chat during an event [which may be of any nature]. The conversations on the group chat are meant to be based around the event and a majority of the conversations on the chat should center based on the event. It is intended to be used in the design of social events where the responses, counter responses and conversations between member of the group lead to better understanding of human communication, coordination and perception under such circumstances.

Scope:

This software will be limited to software engineers and software engineering teams with it's target audience being beta testers domiciled in the group chats which would be subject to these tests. No telemetry would be provided as it is outside the scope of this software.

Document Conventions:

The following would be the standards and conventions/conversions used in this document:

C. *Consumer* — Operator of software *Client* — TEPI Client Object	P. Publisher – see *User Requirements: User Profiles* **S.** Subscriber – see *User Requirements: User Profiles* **U.** *User* — User Account on channel/group

System Overview

System Architecture:

The system would be based on an asynchronous publisher-subscriber design pattern architecture where a user at any point may serve as a publisher and would have to attend to the varying needs of each or no subscriber with each subscriber having a requirement of a particular publisher to satisfy.

Key Features:

The features of this system include the following:

- Scenic observation of an event in a group chat.
- Sending of messages to the group concerning event nuances/details by Users.
- Intelligent response to messages from other Users.
- Below 20% of the conversations on the group chat would be off-topic from the event.
- AI auto generation of messages and responses from participants on the group chat (Users).

User Requirements

User Profiles:

The system would include user roles and these user roles have a single purpose. A User object can have multiple roles at the same time, however each purpose of each role must be carried out by the User object. These roles include:

Role	Purpose
Publisher (pub)	A User account that sends none zero number of messages to a group. This role is will only complete it's life cycle when all it's published messages have been responded to by the actions of another role.
Subscriber (sub)	A User account that responds to messages which have been sent to the group by some publisher(s) on the group. It's life cycle ends when all the messages it's has "subscribed" to satisfy (or respond to) have been replied to by the current Subscriber or other Subscribers. A Subscriber can chose not to respond to a message if it has already been responded to by another Subscriber.

Functional Requirements:

This system should offer a simulation of human interaction with in a group with respect to a particular topic, context or event.

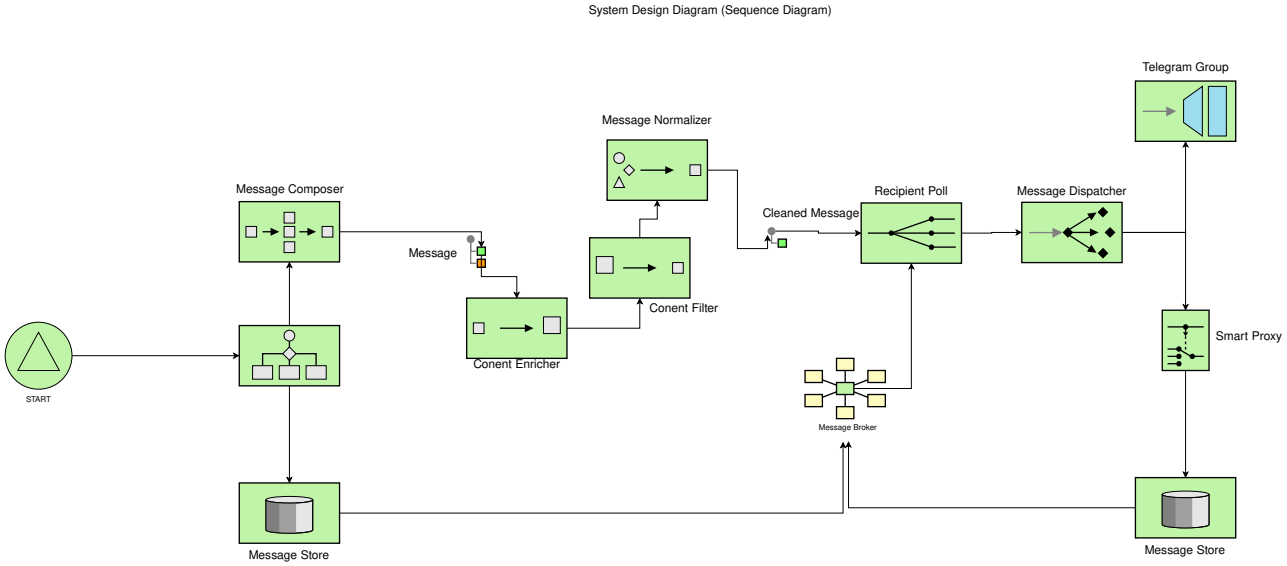
Non-functional Requirements:

The following would describes the non-functional requirements of this system.

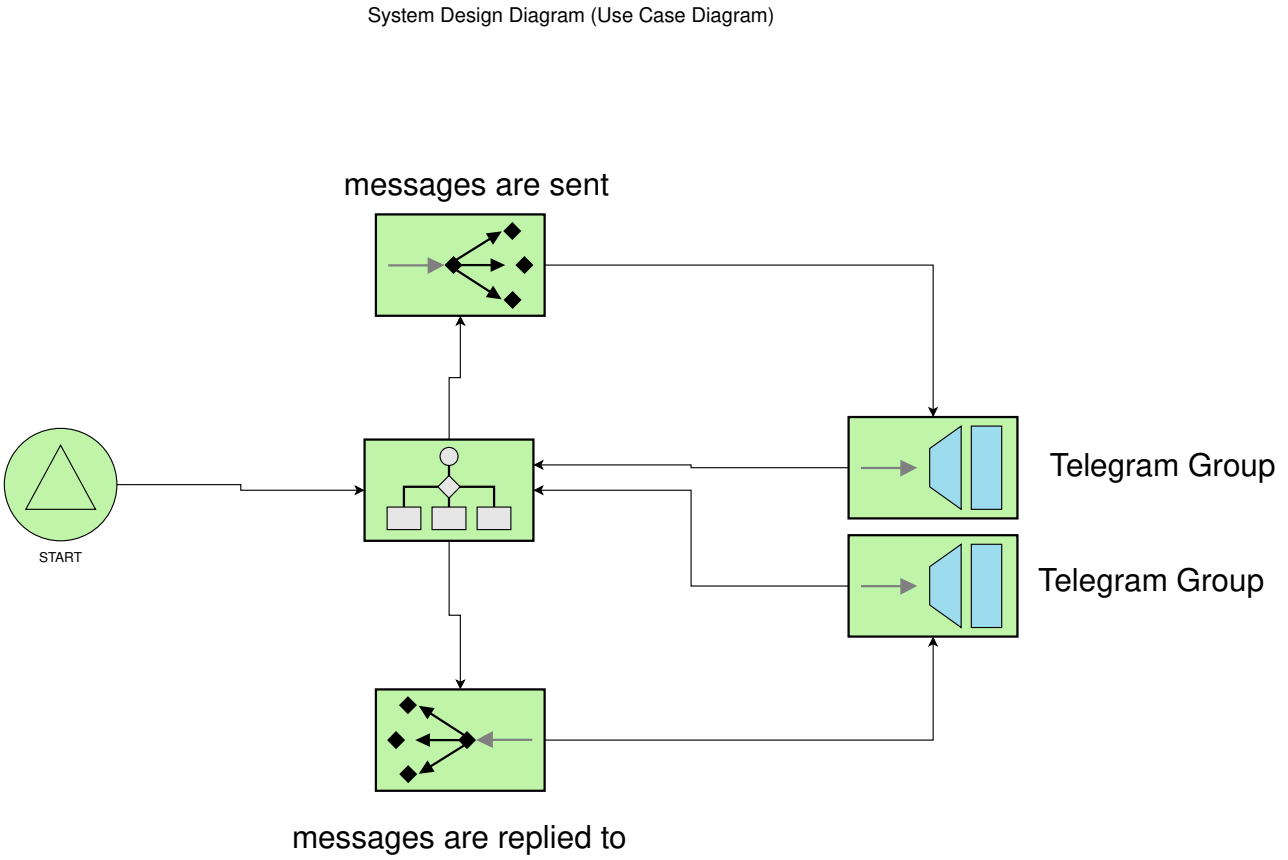
- The performance requirement is that the system should be fault tolerant (i.e. in case the system process restarts, message streams remain accessible and consistent).

System Design

Sequence Diagram:

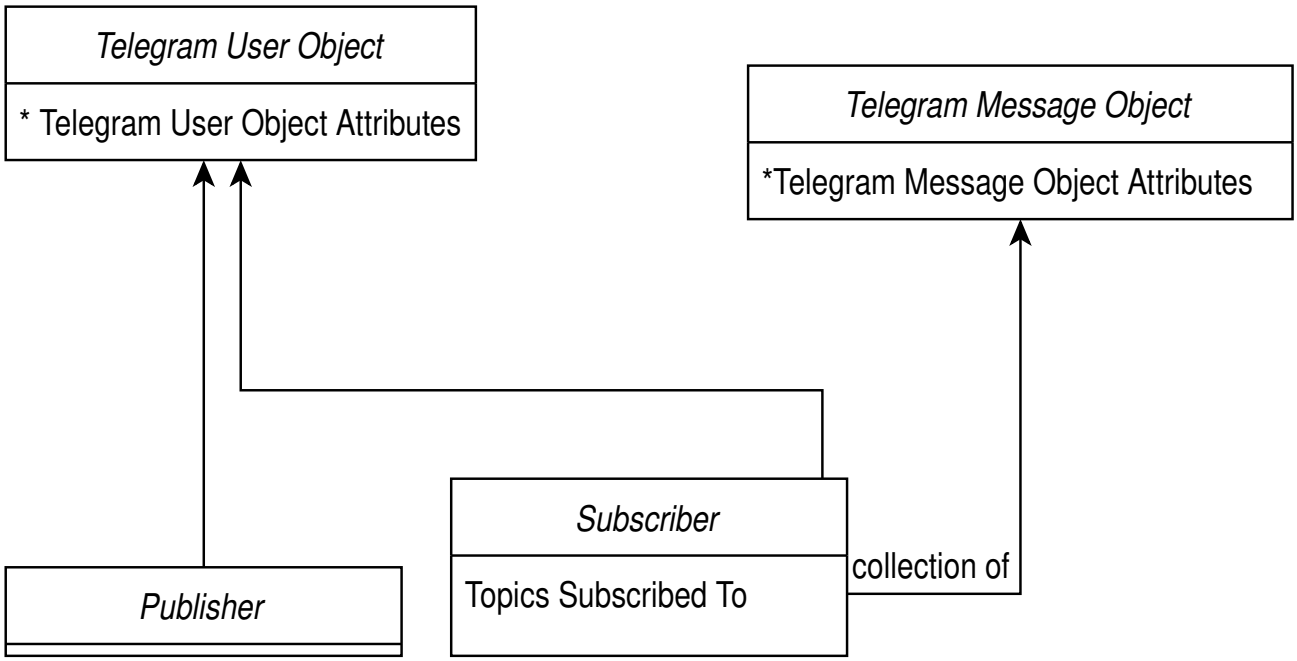


Use Case Diagram:



Class Diagrams:

System Design Diagram (Class Diagram)



Data Design

Data Model:

This application would have 3 major entities:

- The Message Object
- The Message Stream
- The User Object

The Message Object: Represents messages in the group chat.

The Message Stream: This is provided by Apache Kafka Streams API and it would be used for maintaining a message stream which Publishers would publish to and Subscribers subscribe and consume from. The schema of the messages on message stream would first of all follow Apache's expected JSON input: A **key** and **value** would be provided for each message. Each Message's key would be a categorization of what the message could be; either an *initiating_message* or an *ongoing_conversation_message* while the value would be the Message object itself decoded to byte array.

The User Object: Represents users in the group.

Database Design:

The generated messages would not be stored in a database as their records aren't required. This would also reduce operation costs, hence no database would be included in this design doc.

System Implementation

Programming Languages & Frameworks:

Programming languages include:

- Python

Frameworks include:

- FastAPI: *Python network API network.*
- Apache Kafka: *Apache Kafka is a message stream platform.*
- Sphinx: *Documentation tool for Python*

Key Modules:

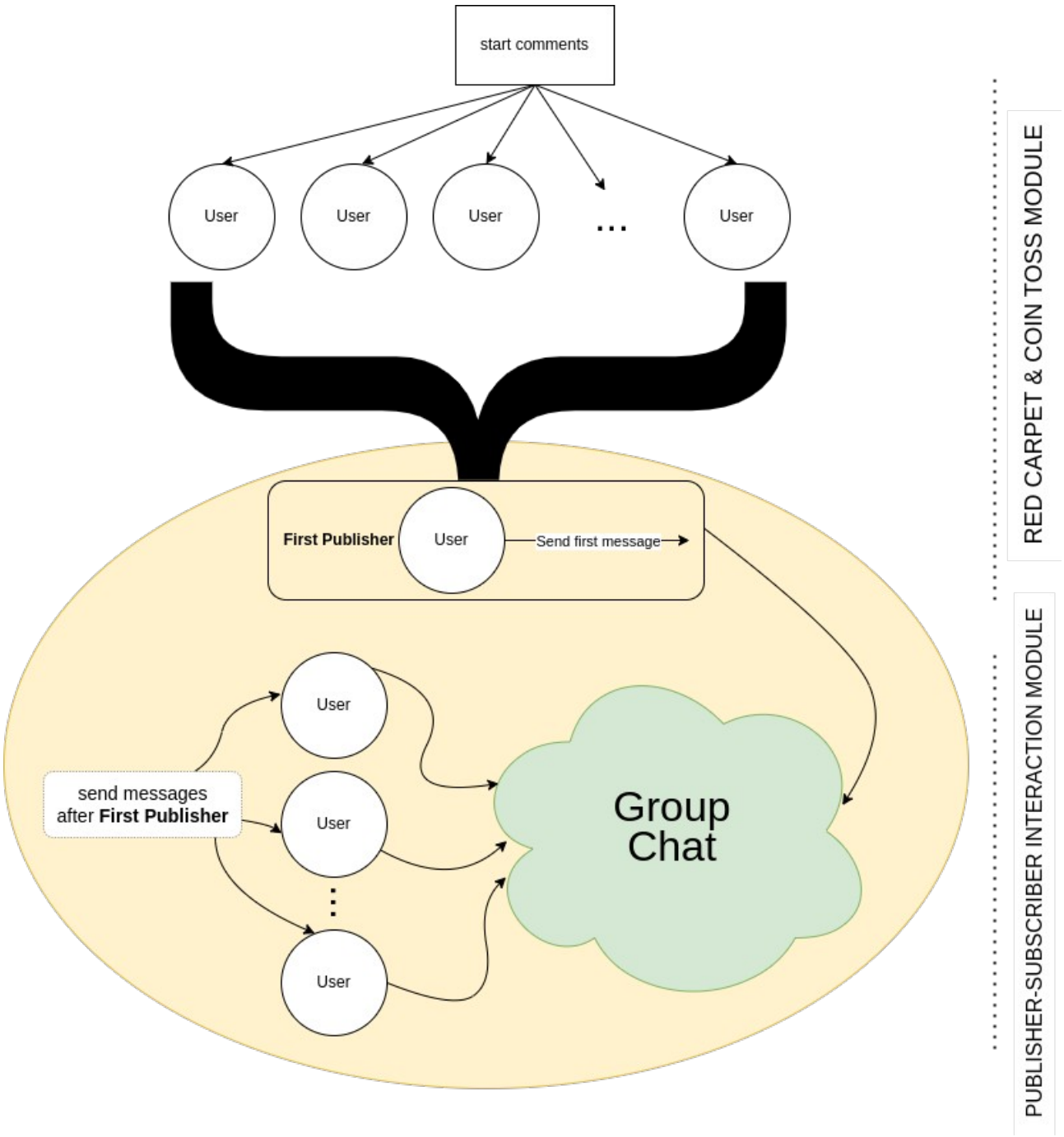
The following are the main modules that would be used in this software:

- Red Carpet & Coin Toss Module
- Publisher Subscriber Interaction Module

Red Carpet & Coin Toss Module: In this module, Publishers and Subscribers are defined, re-defined or relieved of their “Publisher” or “Subscriber” status. In this module as well, the course of action to take is also decided between responding to an already available message within the message stream or posting an new message to the group.

Publisher Subscriber Interaction Module: In this module, a random number of interaction between Subscribers and Publishers are made (i.e. Publisher, Subscriber actions are taken in this module).

Below is a diagram showing a summary of a visual representation of both the *Red Carpet & Coin Toss Module* and *Publisher Subscriber Interaction Module*.



External Libraries & APIs:

For external libraries, see *Programming Languages & Frameworks*. As for external APIs, the following would be utilized:

- FastAPI: *Python network API framework.*
- Apache Kafka Streams API: *API for building streams processing apps.*
- LLM API: *Public LLM API*

Testing And Quality Assurance

Test plan:

The test plan used here will follow Test Driven Development (TDD). None of the libraries'

functionality would be tested however they will be mocked to perform different tests where required.

Test Cases:
Only unit tests would be implemented here since functional tests would require implementing different possible execution scenarios without guaranteeing 100% code coverage.

Quality Assurance:
The following will be the measures to which software quality and reliability is assured:

- Automated Testing: The software will include a test suite which would preferably feature standard library tooling over external libraries and frameworks. This would primarily be built around unittest
- Continuous Integration & Continuous Deployment (CI/CD): GitHub Actions would be utilized for the CI/CD section of this software and will be included in the source code.
- User Acceptance Testing: Beta testers who will stand as beta-client-testers will be involved in user acceptance testing which will check software usability and relative compatibility with real life scenarios.
- Version Control And Change Management: Version control software will be Git and all source code will be hosted on GitHub. Versioning would be of *Semantic Versioning* style.
- Documentation: The source code will be self documenting as all functions classes, and methods will be adequately enriched with appropriate doc strings. Sphinx would be used to generate the PDF documentation from the source code.

Deployment And Maintenance

Deployment Plan:
The steps required to deploy this software in a production environment is outlined below:

1. On push, start automated testing
2. If all automated tests pass, latest commit is published on hosting platform.

Maintenance Plan:
For ongoing software maintenance, bug fixes and updates, semantic version differences would indicate nature of changes made to the source code. Updates and bug fixes can only be triggered by dependency actions/updates, feedback from user acceptance testing and general consumer feedback.

Conclusion

Summary:
The design documentation outlines the development of a software application that mimics human conversations in a group chat during an event. The system follows an asynchronous publisher-subscriber design pattern, where users can act as publishers or subscribers. Publishers send event-related messages to the group, while subscribers respond to those messages. The software aims to simulate human interaction and coordination within the group. It utilizes Python for API interactions, and Apache Kafka for message streaming. Testing follows a Test Driven Development approach, and quality assurance measures include automated testing, continuous integration, user acceptance testing, and documentation generation. Deployment involves automated testing and publishing the latest commit, while maintenance focuses on bug fixes, dependency actions/updates, and consumer/beta-testing feedback.

Future Enhancements:
A future feature for consideration could be the further decoupling of the systems participating in this software where each system can be tasked with performing more complex functions such as smarter multimedia generation, web scraping of popular fact checking sites like Wikipedia for things like quotes or links, e.t.c.

Appendices

References:

- **Wikipedia** — Publish–subscribe pattern