Hello, Friends! Final Project Artifact

Vision Statement

Member names / emails:

1. Tiange Wang / tiange@ucsb.edu
2. Changwei Xu / c_xu@ucsb.edu / Team Leader
3. Jianyu Yu / jianyuyu@ucsb.edu
4. Yifan Yu / yifanyu@ucsb.edu
5. Derek Bang / dkbang@ucsb.edu

This project is a 1-on-1 video chat app that provides translation capabilities. The app will include a main video screen that projects the webcam video feed and several APIs working in the background that provides speech-to-text and text translation functions. The users on both sides first choose the intended language that they would like to use and the language that they wish to translate to, then they can start the video and audio feed where they will be speaking in the language they chose at the beginning and the APIs will process this speech and translate it into the intended language that the user also specified at the beginning. It will be built in Java and the intended platform is PC.

This project has 2 intended user groups. The first group is people hoping to make friends around the world in order to learn their respective cultures and languages and the second group is for business people where communicating in different languages among different teams in a meeting
can present a problem. Today, there are solutions provided for both groups, but they are not suitable for every scenario. For the first group of people, meeting people that are different from them is interesting but it could be difficult for them to communicate. There is online translation software for translating, however, they cannot be conducted in real time which hinders their ability to communicate effectively. For the second group, it is essential for different teams around the world to speak in a language that is understandable so their projects can be carried out without trouble. Even though translators can be provided, they are not always available and it could be difficult to communicate via another person.

Video calls would be possible among people speaking different languages. Nowadays, video calling software such as Facetime, Zoom is able to satisfy the daily video chat needs. This software could satisfy people’s daily requirement of video calling, shorten the distance among people. Employees who work hundreds of miles away from their families could meet easily, despite the limitations of distance. Companies also use video meeting software to boost the efficiency of discussing. With the help of real-time video streaming, remote communication has become swiftly, no longer merely based on language.

However, there are still a lot of video chats strongly dependent on language. For example, How cannot people with different languages communicate with each other fluently? How to convey instructions to foreign people clearly and accurately via video chat? These are the major problems that Video Calling Software need to address and solve.
Therefore, the function of our software is extremely useful and can fill the huge gap in the current Video Calling Software market.

**Timeline:** Research → Basic Structure → Basic Front-End Design→ Basic Front-End Implementation(Using Java swing GUI) → More Research(Mostly on Back-End) → integrate 1-to-1 text chat → integrate video chat (openvidu api) → integrate voice to text (google cloud speech api) → integrate translation(google translate api) → Initial Testing & Polish → additional features such as better front-end, text chat, scribbles to text, watching youtube videos together w/translation, group chat) → Final Testing & Polish → Final Product

---

**Links**

1. Github: https://github.com/TgWang1023/Hello-Friend
2. Travis-ci: https://travis-ci.org/TgWang1023/Hello-Friend
3. Slack: https://hellofriendtalk.slack.com
4. Project demo: https://drive.google.com/file/d/1btoI9HkGKQdb5fDtJSmjEffHGrd6cnV5/view
5. Presentation Slides: https://drive.google.com/open?id=1StdeiimRyIVHLq6rJCRzfd0qlpq6k3zs
System Architecture Overviews:

High-Level Diagram
https://drive.google.com/open?id=1QkBBGFnzFVp-0zCvm9snQ-KIGHtpwFW

Class Diagram
https://drive.google.com/open?id=1UOodFUdQngMRnX9kvMpc88Rqk46k0g8v

Sequence Diagram
https://drive.google.com/open?id=1rtEgQRnTJYkxo8uIHeCpe6ELFxED27RL
https://drive.google.com/open?id=1cfW0MzXIt_xy3QurT_eqZynHTHxa56x
Requirements:

Case 0: Connect to the Server

User story 1: As a user, I can connect to a server so I can interact with users on the server.

Actors: User, Server

Precondition: User has visited the webapp.

Flow of events:

- User clicks “Connect” button at the top of the webpage.

Postcondition: User connects to the server and can create/join room now.

Case 1: Create Chat Room

User story 1: As a user, I can create a chat room so that another user can join my chat and talk to me.

Actors: User, Server, Manager

Precondition: The user has connected to the server, fills and submits the room creation request from.

Flow of events:

Basic Path:

1. Server receives the room-creation request from the user
2. Controller handles the form and calls the create function
3. Controller calls Manager class to create a room with given name
4. If the creation is successful, Manager will return success code to Controller and set up the chat channel
5. The user will receive a success message from Server

Alternative Paths:

1. If the user submits an incomplete form, then the server will return an error message “incomplete form”. (exception case)
2. If the user submits a room name that has been occupied, then the server will return an error message “occupied room name”. (exception case)
Postcondition: The user can now use the channel for chat or he/she has to submit another form again (if creation fails)

Case 2: Join Chat Room with Room Name

User story 2: As a user, I can join a chat room so that I can start a private chat with the room owner.

Actors: User, Server, Manager

Precondition: The user has connected to the server, fills and submits the room join request form.

Flow of events:

Base Path:
1. Server receives the room-joining request from the user
2. Controller handles the form and calls the join function
3. Controller calls Manager class to join a room with given name
4. If the joining is successful, Manager will return success code to Controller and register new user in the chat channel
5. The user will receive a success message from Server

Alternative Paths:
1. If the user submits an incomplete form, then the server will return an error message “incomplete form”. (exception case)
2. If the user submits a room name does not exist, then the server will return an error message “no such room”. (exception case)
3. If the user attempts to join a room that is full (already has two chatters), then the server will return an error message “room is full”. (exception case)

Postcondition: The user can now chat in the channel or he/she has to submit another form again (if joining fails)

Case 3: Language Translation

User story 3: As a user, I can send a message in my preferred language so that the server can recognize and translate my message.
Actors: User 1, User 2, Server
Precondition: User has entered his/her language of choice into the session page and the chat session has started
Flow of events:
Basic Path:
1. User 1 enters message into the text box and submits to the server
2. Server receives the message and sends API request to Google Translate to translate the message into user 2’s preferred language
3. Server receives the translated message from the API and sends the translated message to user 2
4. User 2 receives the translated message
Alternative Paths:
1. User 1 enters empty/invalid message into the text box and attempts to submit to the server
2. Server receives the message and finds the error. Sends error message back to user 1
3. User 1 receives error message
Postcondition:
User 1’s message is translated into the preferred language of user 2 and displayed onto user 2’s interface or message is not sent and error message is displayed.(end-early path)

Case 4: Disconnect Chat Room
User story 4: As a user, I can disconnect from the server so that I will be removed from the chat room.
Actors: User 1, User 2 *(not necessary)*, Server, Manager
Precondition: User is current in a chat session
Flow of events:
1. User clicks on “disconnect” to attempt to disconnect from the chat session and the client stops to receive messages from Server.
2. Server receives the disconnect request and removes the user from the channel.
3. If there is no user in the room, delete the room; otherwise send a system message to notify the other user.

**Postcondition:** The quitting user successfully disconnects from the Server, and the Server will not send messages to the user’s address.

Case 5: Video/Audio Chat

**User story 5:** As a user, I can send video and audio feed to the chat session so that another user in the current chat session can receive my feeds.

**Actors:** User 1, User 2, Server

**Precondition:** User has entered his/her language of choice into the session page and the chat session has started

**Flow of events:**

*Basic Path:*
1. User 1 turns on his/her webcam/microphone to allow video and audio transmission.
2. Video feed is passed directly from front-end into the OpenVidu API to establish RTC connection.
3. Once the connection between user 1 and user 2 has been established, feeds are transferred automatically
4. User 2 receives video/audio feed from user 1

*Alternative Paths:*
1. User 1 turns on his/her webcam/microphone to allow video and audio transmission.
2. Video feed is passed directly from front-end into the OpenVidu API and attempts to establish RTC connection but fails.
3. Error status is returned to the front-end and displayed on user 1’s interface.

**Postcondition:** User 2 receives real-time video/audio feed from user 1 or error message is displayed on user 1’s interface. (end-early path)
Case 6: Incomplete Input

**User story 6:** As a user, I can receive the error message in my preferred language if I didn’t fill all entries in the form.

**Actors:** User, Server

**Precondition:** User has entered his/her language of choice into the room form but not all entries

**Flow of events:**

1. User clicks on “Create” or “Join” button.
2. Server recognizes the incompleteness form and send back error message in the user's selected language.
3. The user will then receive the error message “Please fill all entries in the form.”

**Postcondition:** User knows the error and need to refill the form.

Case 7: Connect Message

**User story 7:** As a user, I can receive the system message in my preferred language if a user joins my room.

**Actors:** User 1, User 2, Server

**Precondition:** User 1 has created the room and waiting for other people to join

**Flow of events:**

1. User 2 joins the room with entering the room name.
2. Server recognizes the availability of the room and enrolls User 2 into the room.
3. Server sends a message to User 1: “A user has joined the room.” in the preferred language of User 1.

**Postcondition:** User 1 knows that User 2 has joined the room.
Case 8: Disconnect Message

**User story 8:** As a user, I can receive the system message in my preferred language if another user quits the room.

**Actors:** User 1, User 2, Server

**Precondition:** User 1 and User 2 are in the same room

**Flow of events:**
1. User 2 clicks “disconnect” button.
2. Server receives the disconnect request from User 2 and delete all information of User 2.
3. Server sends a message to User 1: “Another user has disconnected.” in the preferred language of User 1.

**Postcondition:** User 1 knows that User 2 has quit the room.

Case 9: System Message

**User story 9:** As a server I can send system message with a single function call so that the design is said to be highly encapsulated.

**Actors:** Server

**Precondition:** Server knows a status code to send and needs to compose a corresponding error message.

**Flow of events:**
1. Server asks the Manager with the error code, destination language and call Manager.systemMessage().
2. Manager will return a system message and translates into the desired language.
3. Server can then send the returned message to the client directly.

**Postcondition:** Server deals as little logic as possible, achieving well-encapsulated design.
Retrospective info:

Sprint 1
1. Build a rough structure of the project, enable users to chat within a channel.
2. Set connect/disconnect mechanism to deploy SockJS.
3. Create templates of the backend system.
4. Deploy Google Translation API.

Sprint 2
1. Create UI (front end) for the web app.
2. Optimize mechanism of backend system: error handling, message translation, merge front end and back end.
3. Test for back end templates.
4. Deploy Google Speech to Text API.
5. Begin researching on voice recording.

Sprint 3
1. Optimize user experience: join/quit message, translate all system messages.
2. Make back end system more encapsulated and working coherently.
3. Test for front end and back end.

Challenges Faced/Overcome:
1. Spring Boot only supports channel broadcast instead of private message. So we give each user a unique ID and a private channel so that the user can only receive messages from his/her room.
2. The Google Cloud APIs are messy and needs to import many libraries. So we build an individual package for all Google Cloud APIs, making use of them easy.

3. A lot of process in our app are repeated so that if we want to modify details, we need to modify many places at once. So we merges all repeated logic into a static class Manager. Then we only need to modify a single method of Manager in order to make changes to the logic.

---

**Missing/Remaining Features or Functionality:**

Our final demo version of the project is merely one part of the original idea. Due to technical difficulties we did not successfully implement these other features, but our effort put into researching these functionalities can been seen in our branches on GitHub. The final version of our project includes a 1-to-1 text chat that provides translation capability and a room system without registering accounts, which is our goal in sprint 2. In the second half of sprint 2 and sprint 3, we mainly focused on implementing video and audio features, which we did not successfully implemented due to a variety of reasons. (For more details on our failed implementation of video and audio, checkout branches “video-chat” and “audio” on our GitHub page.)

For video, our team tried to use the OpenVidu API to integrate video chat into our project. Unfortunately, we encountered 2 major issues, which are incompatible structure and security breaches. For incompatible structure, the API uses an MVC based project for the implementation, which cannot be used on our single-page project. As for security breaches, the OpenVidu API uses docker to run WebRTC real time connection video server, which is blocked by the web as malicious connection and we did not find a way to overcome this issue. We also tried other APIs, however all of them are deprecated and thus cannot be implemented into our project.
For audio, we also wrote the corresponding code to try to support the “press a button and start recording” feature. The API that we were going to connect was the Google Speech-to-Text API, which takes 2 major forms of audio input: file and microphone. As first we tried microphone, but the issue with microphone is that it is almost impossible to send real time audio data and transcribe it into text, which is why we give up on that. As for file, we spent a lot of time working on converting voices into a .wav file, which can be used as an array of bytes and pass directly into the API. Unfortunately, doing that would require us to change the structure of our entire project, which is why we also give up on this feature as we don’t have enough time.

As for additional features that we want to implement, a chat room that supports multiple users would definitely be the number one priority. The current system is based client-server structure which hold information about all on-going chat rooms and users. It is difficult and inefficient to maintain such a large system using a single server. In order to support multiple users, one thing that we could do is to change our implementation into a peer-to-peer structure that is scalable based on the number of users. Another feature that we wanted to implement is the ability to send pictures. This should be achieved quite easily as we only have to process the image as bytes and send directly to the other user. Doing so, however, would require us to adjust our front-end to support image display. Responsive design for a mobile-friendly design also need to be considered.

__________________________

**Github Commit Timeline:**

Burndown Chart and Commit Timeline:
https://docs.google.com/spreadsheets/d/15tzTguhGWPlZ93g8ilgVywRzoRURDCSMjldRjRTpFs/edit?usp=sharing
Test Files:

Back End Test (location: src/test/java/edu/ucsb/cs48s19/)
1. ManagerTest: test for class Manager
2. AdvancedMessageTest: test for class AdvancedMessage
3. JoinTest: test for class JoinRequest
4. MessageTest: test for class Message
5. PairTest: test for class Pair
6. RoomTest: test for class Room
7. UserTest: test for class User

Front End Test (location: tools/)
Details are included in tools/Readme.txt and README.md.

Appendix:

List of technologies employed: Java, HTML5, CSS3, Javascript(ES6), Apache Maven, Spring Framework, Google Translate API, Google speech-to-text API, OpenVidu API, Git, travis-ci