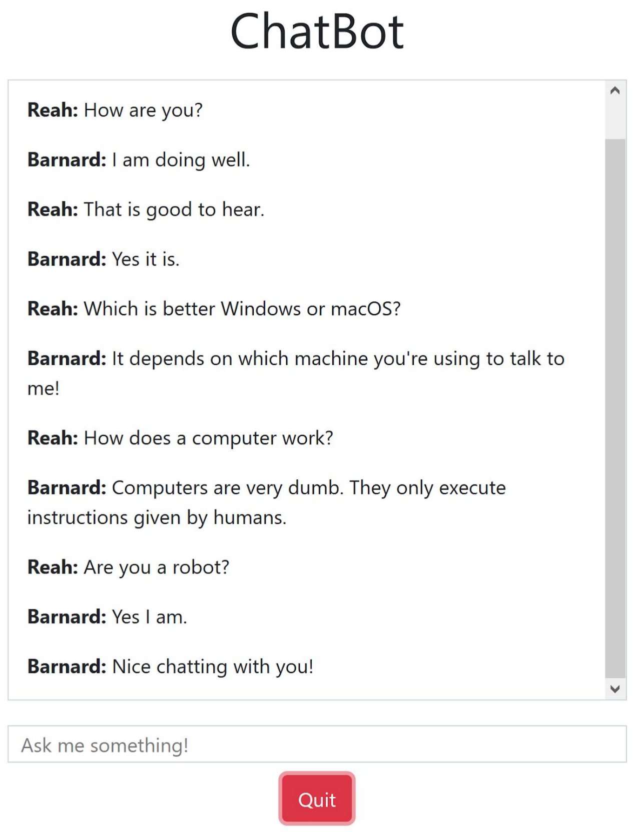
Design Report

Anthony Cheng and Reah Rajmangal

The project that we implemented was: “A simple chat bot that responds to simple commands such as “the current time”, “4 + 5”, etc….”

We implemented this in Python using primarily the websockets and a machine learning dialog engine called Chatterbot.



Reah: How are you?

Barnard: I am doing well.

Reah: That is good to hear.

Barnard: Yes it is.

Reah: Which is better Windows or macOS?

Barnard: It depends on which machine you're using to talk to me!

Reah: How does a computer work?

Barnard: Computers are very dumb. They only execute instructions given by humans.

Reah: Are you a robot?

Barnard: Yes I am.

Barnard: Nice chatting with you!

Install these python modules:

- ❏ asyncio
- ❏ websockets
- ❏ chatterbot
- ❏ webbrowser

After installing dependencies, run server.py and wait for a few seconds for the chatbot to be trained. A browser should open up with the chatbot HTML page and you can begin chatting with Barnard!

Introduction

We planned to use a server (socket) that would accept new connections (basically one user) and broadcast messages. The chat bot will be in the server and will create replies to messages based on the input of the client. We expect that the user will ask basic questions such as “what is…[the date, time]” and can conversate with our bot Barnard, who is not the brightest.

After running server.py, bot will log their training on the console and then open up a page in browser titled “Chatbot” where you can type you name and begin chatting.

Questions you can ask Barnard:

- How are you?
- Are you a robot?
- What time is it?
- What is number [*, +, /, -] number

After closing the browser/page, the websocket connection is automatically closed and the program will exit.

Implementation

The first part of implementation consisted of creating a connection between a client and server where all the data is processed. We initially wanted to use the socket API in conjunction with the Tkinter GUI, but chose to work with Websockets instead to create a simple HTML page for our GUI. We moved to this approach because creating HTML pages and running a small client-side code in javascript was simpler.

Based off this basic example, we first created a server using the websockets serve() function to which it takes a handler coroutine (aysynchronous function that pauses if not executing anything), localhost, and local host port, in this case we kept with port 5678. The serve function creates, starts, and returns a websockets server. The handler coroutine consists of all the processing between the client and the server. We then allow that code to be run in an infinite loop until the handler returns.

Then we created an interface for the client to send data. The simple HTML page would hold the chat messages and allow the client to submit new messages to be handled by the server. In the script portion of the HTML page, we created a websocket that is binded to the localhost port as coded in the server. The client can then send messages back to the server and the server receives those messages. In the coroutine chat () function, the server receives the first message as the name. Then it will repeatedly wait for new incoming messages from the client. It then processes this data using Chatterbot and formats it along with a new response and sends it to the client.

As we can see, there are a lot of similarities between the socket API and websockets. They both require creating a connection and binding to that connection to send data back and forth. One important difference however was that there was explicit closing of sockets after the client was finished connecting with the server and as a result this would throw a long error to the server saying that a client websocket had been closed and keep the server running.

Because the only time the server would be running is when the only client is connected to it, we simply modified the websocket's normal behavior to exit the server code once the client has disconnected. In order to do this, we wrapped the coroutine code within a try/except block and caught the exception if the client exited out the page to which the server would exit out as well so that all resources could be freed. We also allowed the client to quit the chat and explicitly close the websocket.

The last issue regarding the websockets had to do with the conjunction between the chatbot itself and the websockets. We wanted to make it the easiest for users to start the chat so rather than explicitly running the code, opening the page, and waiting for the chatbot to be trained, we just allow the user to run the server and just wait for the training to complete and have the page open up automatically -- hence the need for the webbrowsers module. Chatbot needed to be global because the coroutine handler could not pass the chat explicitly.

The second step was implementing a chatbot using Chatterbot. Using the Chatterbot documentation I was able to create a function to create a Chatbot. It was pretty simple to do this because the documentation lays out how to do it. Basically you first define the function and then you make an instance of the ChatBot. Next in the instance declaration you have to let the bot know what types of responses it is capable of replying too. Since we wanted a time, math and regular chat bot we used the "chatterbot.logic.MathematicalEvaluation" and the "chatterbot.logic.TimeLogicAdapter" properties of chatterbot. Also, I implemented the "chatterbot.logic.BestMatch" because that allows the ChatBot to determine the best response instead of sometimes being a little random. I set the threshold to 0.65 for 65% which means if the bot is less than 65% confident then it would just use the default response I created. The ChatBot trains when you run server.py then when you open the ChatBot you are ready to go. I set the chatbot with the trainer using the chatterbot.trainers built in functions and in english but ChatBot does support other languages. Lastly, I returned the chatbot so it can be used in chat. In chat there is a global chat bot that is created and when it is initialized after the server start it creates the ChatBot and Barnard is yours for the taking.

Some issues I had was that at first I did not know chatterbot was a module available to download. At first I wanted to hardcode responses but realized that would not be viable. Shortly after, Reah explained to me what chatterbot is and I realized that it made it really simple to have a functioning machine learning type chat bot. It was also annoying to train the ChatBot and I had an issue with figuring out how to get the default response to render. Sometimes when I typed “hi” it would comment back the time instead of a proper response like “hello”. In general, the responses returned fit and the ChatBot was very fun to make. Hope to work on more networking projects on my own time and expand on these things we learned.

References

- https://websockets.readthedocs.io/en/stable/intro.html -- how to use Websockets API
- https://websockets.readthedocs.io/en/stable/api.html -- different Websocket APIs functions
- https://hackernoon.com/a-simple-introduction-to-pythons-asyncio-595d9c9ecf8c -- understanding what asyncio is and how it works in conjunction with websockets
- https://github.com/gunthercox/ChatterBot -- an introduction to Chatterbot
- https://chatterbot.readthedocs.io/en/stable/ -- further documentation on Chatterbot