Report for lab 1

a81060bs

March 2023

1 Protocol

PROTOCOL:

- THE PROGRAM RUNS IN THE TERMINAL.
- THERE ARE EITHER 1 OR 0 MESSAGES ON THE SERVER at any time.
- THERE IS ONLY ONE KEY IN THE SERVER, IO 'sender'.
- INITIALLY, THE SERVER IS EMPTY. – otherwise, manually clear the server.
- THE SERVER FOLLOWS THE FIRST-COME FIRST-SERVE SCHEMA.
- SO IF A USER RUNS THE APPLICATION FIRST - HE IS A SENDER AND **MUST** SEND A MESSAGE.
- AFTER THE 'sender' HAS SENT THE MESSAGE THE SECOND USER CAN LOG IN.
- THE SECOND USER AUTOMATICALLY IDENTIFIES AS A 'receiver' AND RECEIVES THE MESSAGE INSTANTLY.
- NOW, conversation() STARTS. WHICH FOLLOWS THIS SCHEMA
  - REPEAT UNTIL SOMEBODY LEAVES THE MESSENGER.
    - ∗ SEND MESSAGE
    - ∗ WAIT FOR RESPONSE
    - ∗ RECEIVE THE MESSAGE

2 Restrictions

There are lots of restrictions and bugs in my program solving which would require a lot more time than what is required by the laboratory.

The first and fairly obvious restriction is that the server must be empty before the first user starts the program. Otherwise, user1 and user2 will be automatically assigned a value of a ‘receiver’, this will cause a deadlock as none of the users will be able to send a message. I have tried to reduce the probability of this happening by introducing a function ‘leaveConversation()’, which tries to clean up the server if the user has decided to leave the chat, by doing so keeping the server clear for future use.

Yet this might not always work, for example, if a user would hit enter instead of exiting the program, he would be required to send another message, and if the second user has already left - this might cause the first user to remain in ’waiting’ state forever - creating a deadlock.

Another limitation of the protocol arises when two users start the program simultaneously. In this scenario, both users are assigned as ’sender’, leading to a potential deadlock.

3 Synchronisation

To synchronize messages between users, the server repeatedly fetches for new messages. This is done through the fetchOrWaitForResponse() function, which is called every time a message is sent. What this function does is it fetches a message at the moment when it has been sent by a user, io fetches the message which was sent by the user - this is illustrated in line 2.. It then enters a loop that waits until the message in the server changes 2.. This is accomplished by continuously checking the server for any changes every 0.1 seconds. Once a new message is received, the loop is exited, and the new message is printed to the user. If no new message is received, the function continues to wait for a response. If a deadlock occurs the whole system is rebooted, by cleaning the server and starting the function start().

```
1.  def fetchOrWaitForResponse():
2.       newMessage = server['sender']
3.       while newMessage == server['sender']:
4.              time.sleep(0.1)
5.              waitForResponse()
6.                continue
7.      print("\n********** new Message! ********** \n")
8.      print(server['sender']

#Helper function.
def waitForResponse():
    waitMessage = "Waiting for response..."
    print(waitMessage, end='\r', flush=True)
```

4 Appendix - What does each function do?

Out of the 9 functions present in the program, only 3 of them are responsible for handling the conversation between the users. The remaining 6 functions serve as helper functions, assisting the main functions in performing specific tasks.

The list of functions:

1. waitForResponse() – just a helper function that prints ' Waiting for response'
2. fetchOrWaitForResponse() - explained above
3. leaveConversation() - a helper function, it is run when users are in the conversation state, it is used to exit the conversation and clear the server.
4. The Start() function serves to categorize users based on whether or not there are any messages stored in the server. If there are no messages, the user is classified as a 'sender' and must send a message. Alternatively, if there are messages, the user is categorized as a 'receiver' and must wait for a message to be received. Additionally, the Start() function initializes the conversation() function
5. The Conversation() function begins after each user has sent a message, and it involves a repetitive process of exchanging messages, whereby one user sends a message, and the other receives and responds to it. Essentially, it is a loop that consists of sending and receiving messages. It is broken when leaveConversation() is runned. Those functions do exactly what the name suggests.
6. sendMessage(username)
7. getUsers()
8. solveDeadlock()

5 Running the file

1. Make sure the server is clear.
2. If it is not clear run server.clear().
3. Run the program by using:
   - python3 iamclient_a81060bs.py
4. Send the message.
5. Run the program on a different machine/terminal.
6. Send the message.
7. Now you should be free to text each other.