**Whiteboard application**

**Purpose:**

Main goal behind developing this application is to provide a stable platform for communication among users. A whiteboard is to be implemented where a user can be registered and upload a topic, any user can send a message under selected topic and all of those who have subscribed to that topic will have the option to view these messages. Record is to be kept for who is the creator of any topic or message. Only creator of the topic will have the choice of deleting it and along that all messages under that topic will be deleted.

**Design Choices and Pattern:**

Higher level design pattern used in this application is client-server architecture with all resources available to server and client is going to interact with the server. Client in this are the users of this platform whereas server manages all resources and permissions.

Implementation of the program is done in c language. Although there isn’t any choice of making classes in this language but while developing the software, design to manage server resources is kept as close to object-oriented as it could be.

To achieve this four structs are available as header files, along with functions for each struct in separate files. Whiteboard object is of the upmost layer comprising of maximum number of users and topics, number of available users and topics, and list of all users and topics with each entry pointing to corresponding object. Data Variables for whiteboard object is listed below.

typedef struct {  
 Topic \*\* topicList;  
 User \*\* userList;  
 int currentTopics;  
 int currentUsers;  
 int maxTopics;  
 int maxUsers;  
} Whiteboard;

Users object will only have a username and password.

typedef struct{  
 char username[64];  
 char password[64];  
}User;

As topic has its creator, list of messages, list of subscribers and a name so it has the following datatypes.

typedef struct {  
 char topicName[64];  
 Message \*\* messageList;  
 User \* owner;  
 User \*\* subscriberList;  
 int currentMessages, currentSubscribers;  
 int maxMessages, maxSubscribers;  
} Topic;

Last object to be describes is of message. All messages will keep record of their owner/creator along with message status and the message text.

typedef struct{  
 int messageId;  
 char messageText[1024];  
 char status[16];  
 User \* owner;  
}Message;

All this data is only accessible to the server and server is responsible for processing it and passing the information to the user.

**Interaction of Modules:**

There are four major modules as described earlier also referred to as “objects” in this document. Access to data is strictly managed and only given to those modules that need it.

Server will only interact with Whiteboard and Topic module to make it simpler and to increase security by limiting the data access.

Modules interact with each other using available functions that will be described below. Whiteboard module has following functionalities.

* + initWhiteboard(Whiteboard \* whiteboard, int maxTopics, int maxUsers )

This function is used to initialize a whiteboard object.

* + addUser(Whiteboard \* whiteboard, char\* username, char\* password)

This function adds a user with specific username and password to the whiteboard initialized earlier

* + authenticate(Whiteboard\* whiteboard, char \* username, char\* password)

To authenticate if a user is giving correct credentials this function is used. It takes “username” and “password” as its parameters and compare from the database whether provided username corresponds with the password or not. In case if these don’t match null is returned else the User object is given to make any changes from that user.

* + addTopic(Whiteboard \* whiteboard, char\* topicName, User \* owner)

This adds a topic with topic name as provided and “owner” as its creator in the available whiteboard. Returns –1 if the topic is not added else 0.

* + getMessageTopic(Whiteboard\* whiteboard, int messageId, User\* user)

If message is to be extracted using message id which is managed automatically, this function is used. It returns the pointer to message object corresponding to the provided id only if “user” is the creator or subscriber of that Topic.

* + replyMessage(Whiteboard\* whiteboard, char\* reply, User\* user, int messageId)

This is used to add a message under any topic using message id. Reply represents that it will be added in thread of messages ( or under topic ) containing the message with specified id.

* + getTopic(Whiteboard\* whiteboard, char\* topicName)

This function is used to get Topic object with corresponding topic name.

* + deleteTopic(Whiteboard\* whiteboard, char\* topicName, User \* user)

As clear from its name this function deletes a topic if the user provided is its creator.

* + getMessageList(Whiteboard\* whiteboard, Topic\* topic)

This function is used to get all messages under a specified topic.

* + getAvailableTopicList(Whiteboard\* whiteboard)

This function gives all available topics.

Now moving to functionalities of our next important module that is “Topic”.

* + initTopic(User \* owner, char\* topicName, int maxMessages, int maxSubscribers )

This function initializes a topic object.

* + subscribeUser(Topic\* topic,User\* user)

If a user wants to subscribe to a specific topic in order to reply to that topic this function is used.

* + isSubscribed(Topic\* topic,User\* user)

For limiting the access to topics this function is used. It checks whether or not specified user is subscribed to the “topic”.

* + addMessage(Topic\* topic, char\* message, User\* user)

To add a message under specified topic this function is used. It also confirms that user is either subscribed to the topic or is its creator.

* + getMessage(Topic\* topic, int messageId)

This simply returns the message corresponding to the given message id under specified topic.

* + deleteTopicDetails(Topic \* topic)

It is used to delete all messages under topic specified in parameter.

All of these functions can only be called through Server or within the Module. Through selective order of these function calls any specified task can be achieved by server.

**Program Flow:**

To understand the program flow it is necessary to keep in mind that this application is based on client-server architecture. Client represents users. Any user starting the application, has to authenticate himself first by providing username and password to the server and server then replies with whether or not this user is allowed to proceed. After the authentication user is provided with 10 possible options. For each selected option server performs the task and then returns confirmation or some data to the user. These options are as follows:

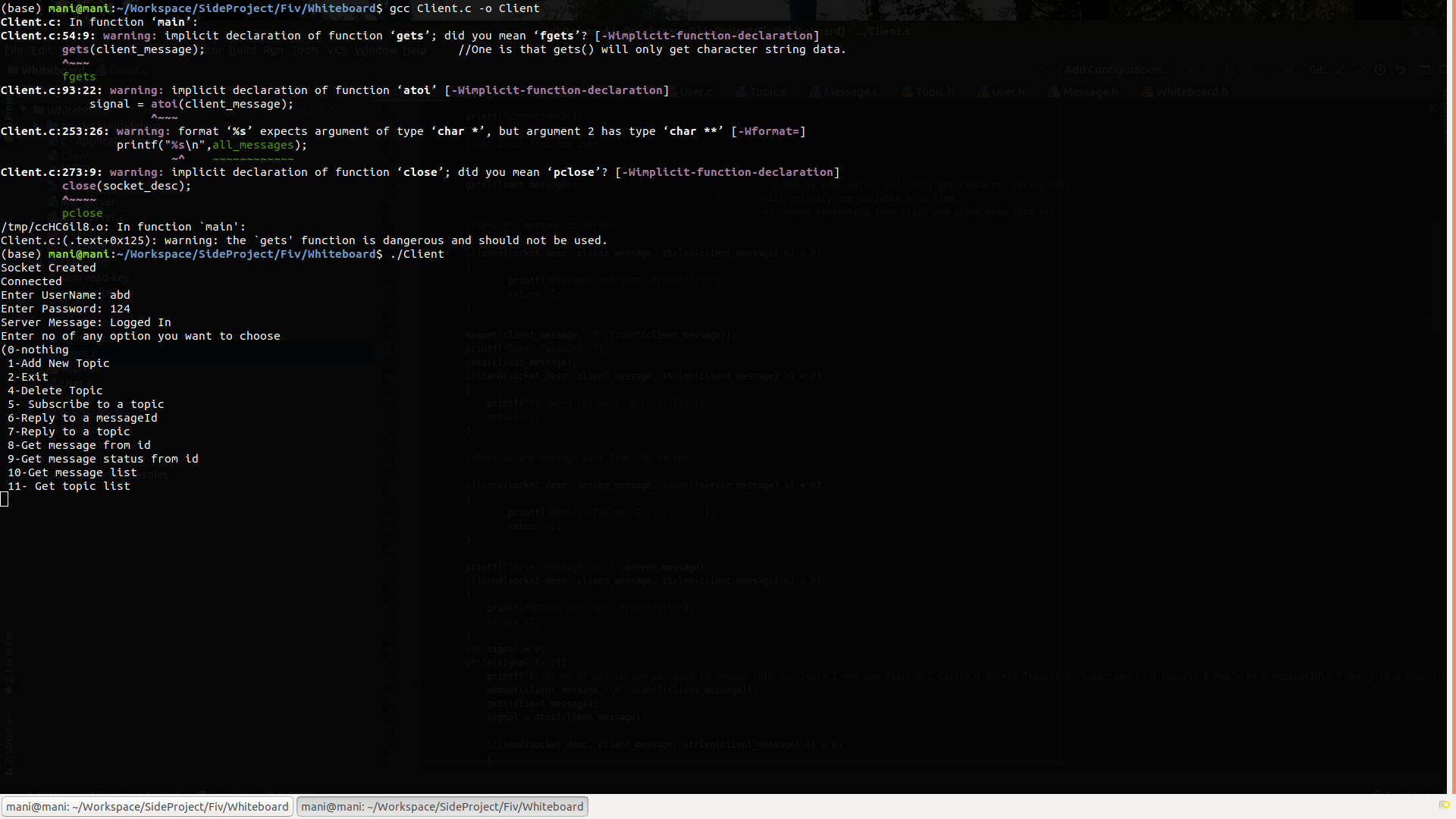
* + Adding a topic
  + Deleting a topic
  + Subscribing to a topic
  + Reply to a topic by providing message id of any message under that topic
  + Reply to a topic by providing topic name
  + Get message by providing message id
  + Get message status by providing message id
  + Get list of all messages under specific topic
  + Get the list of all topics

Nomenclature for functions and other variables is chosen, keeping in mind its functionality and purpose to make code self-explanatory.

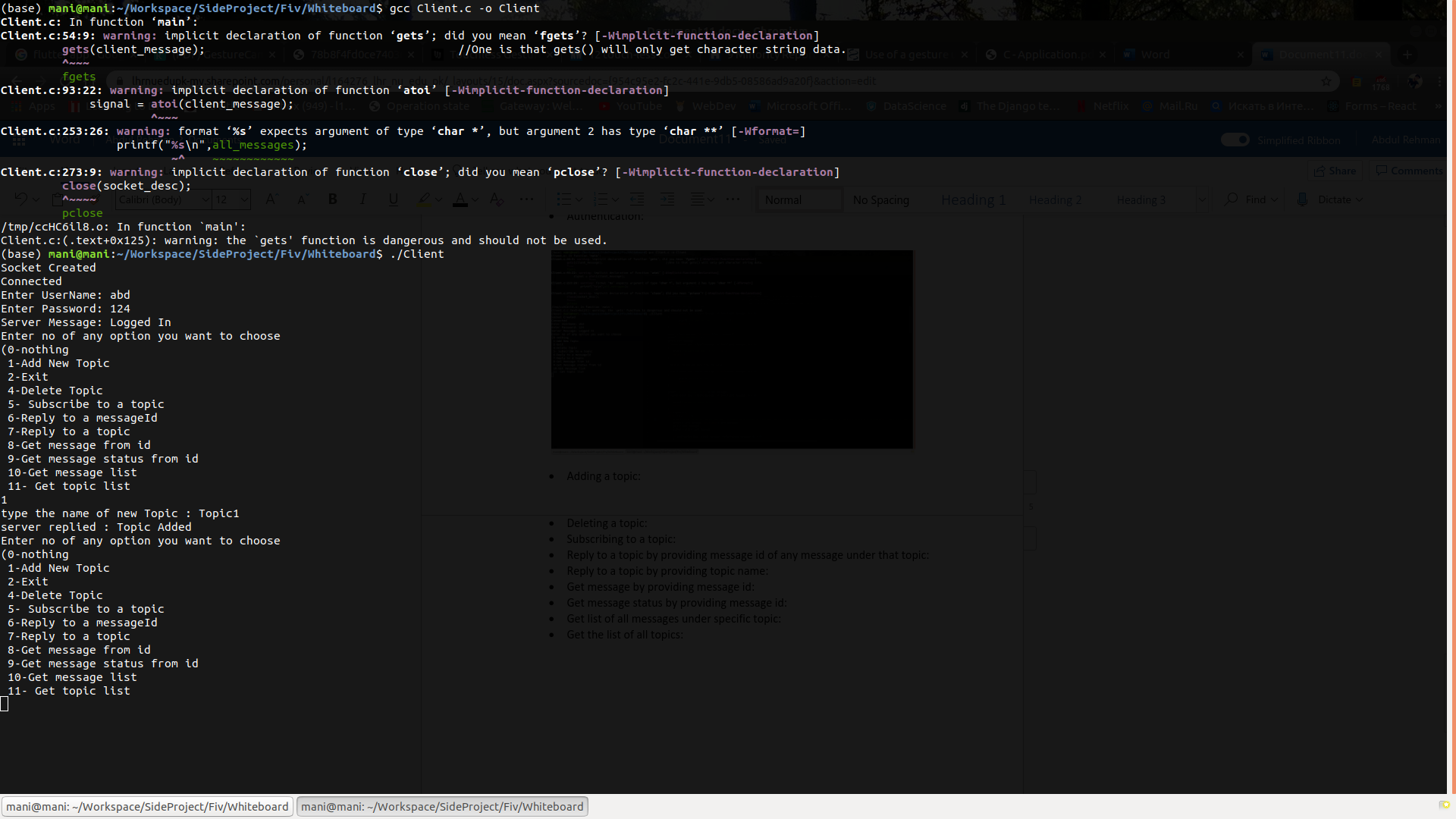
**Test Cases:**

Test case is provided for authentication and some options listed above.

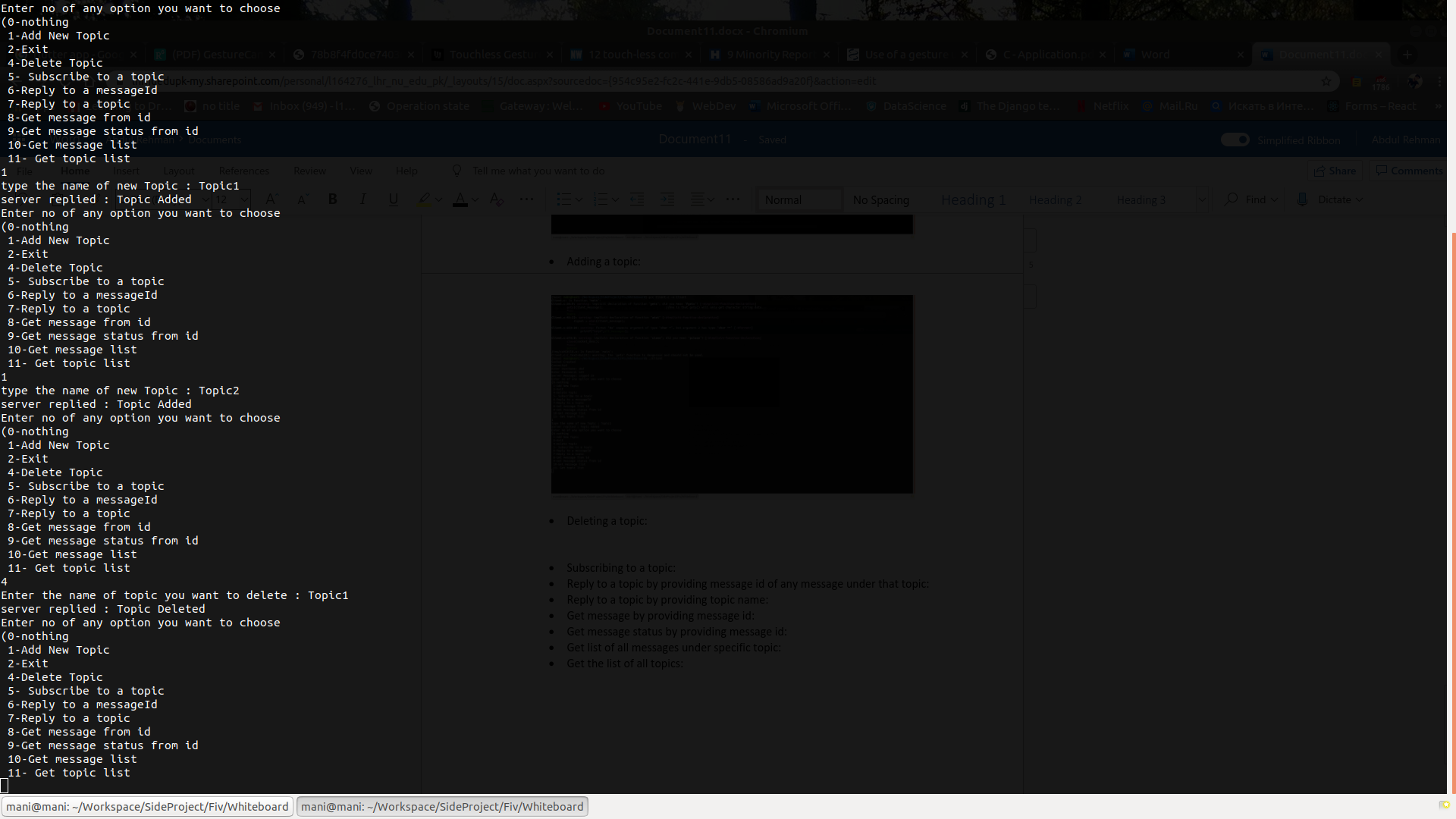
* + Authentication:



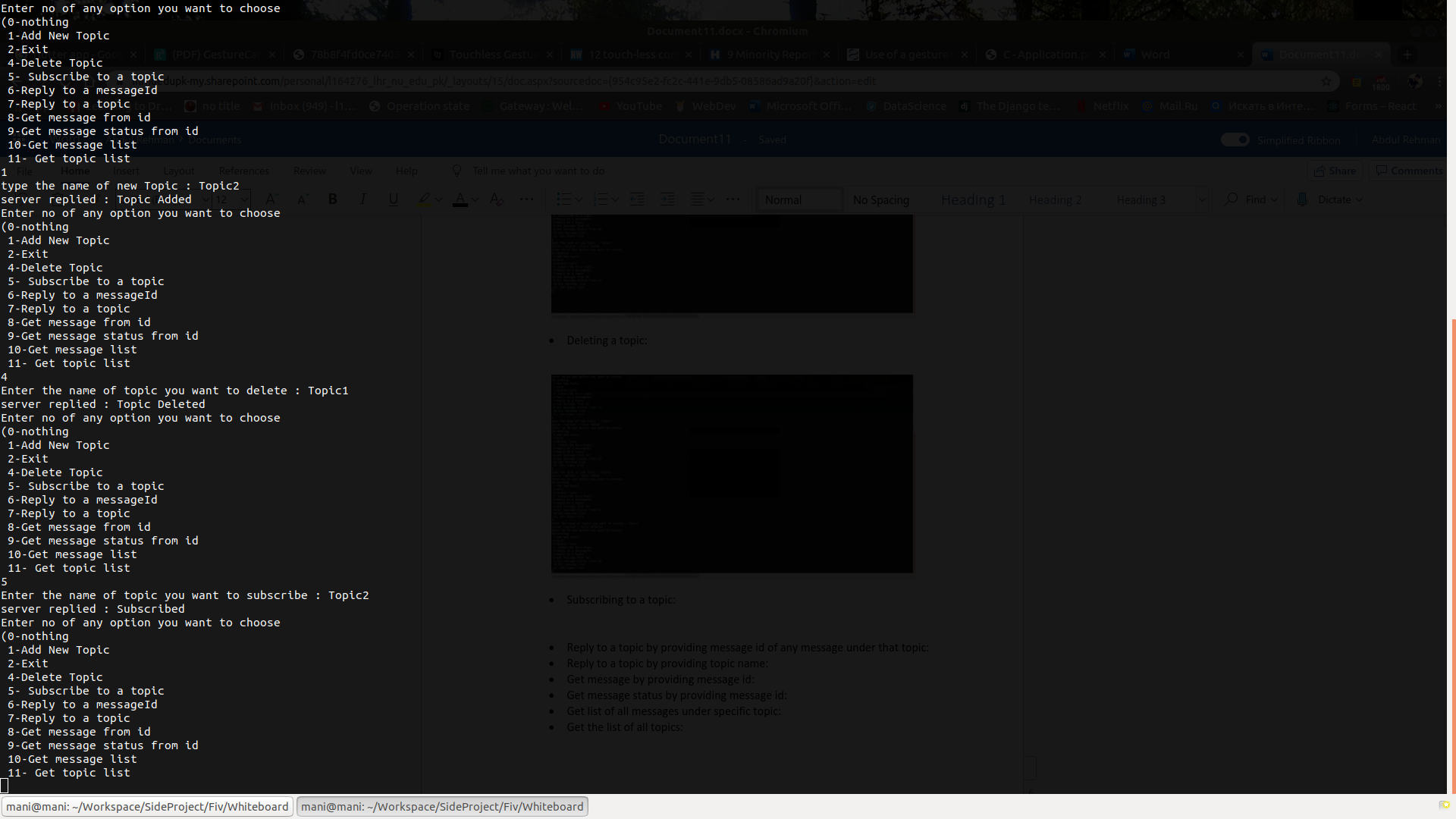
* + Adding a topic:



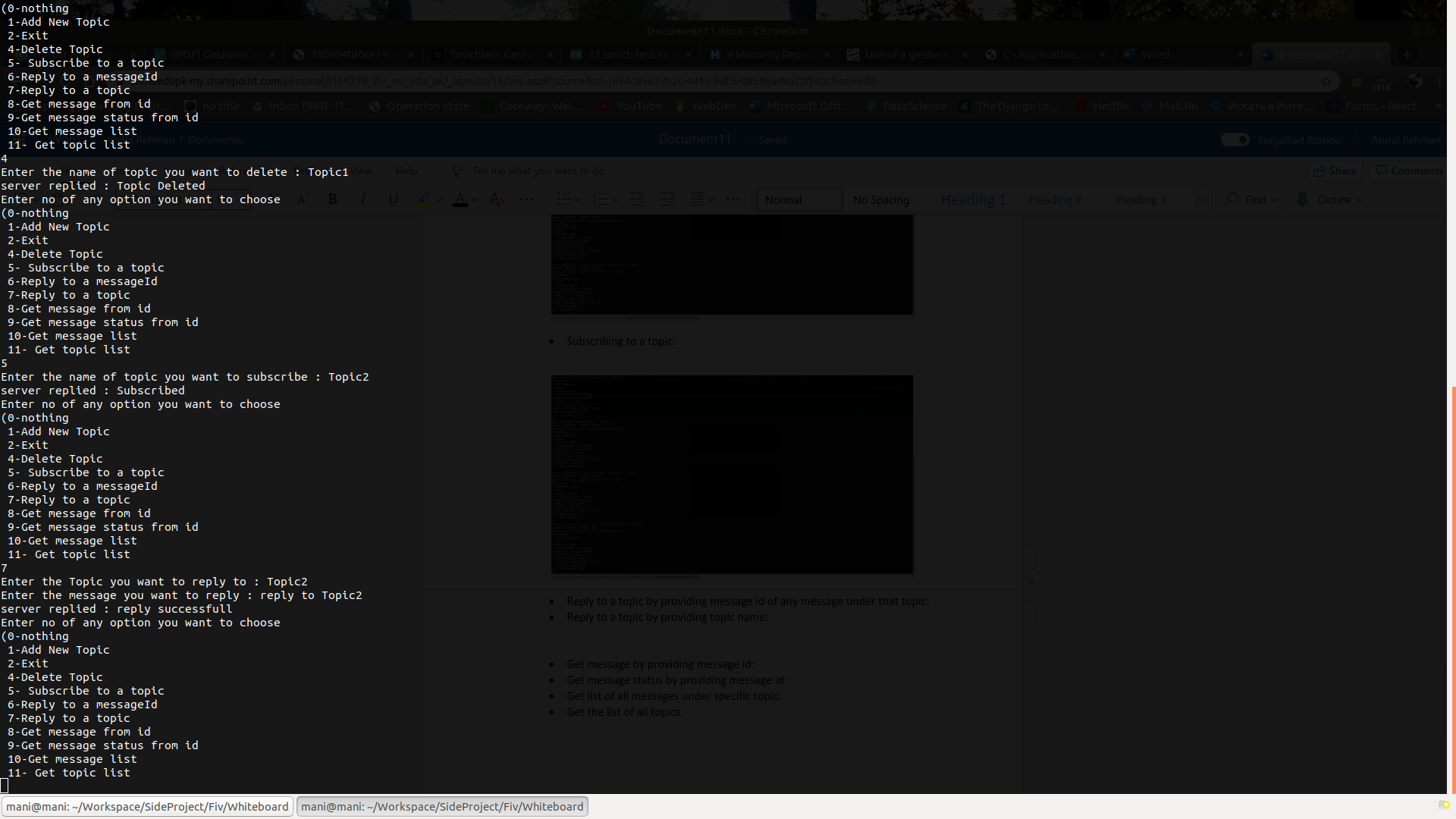
* + Deleting a topic:



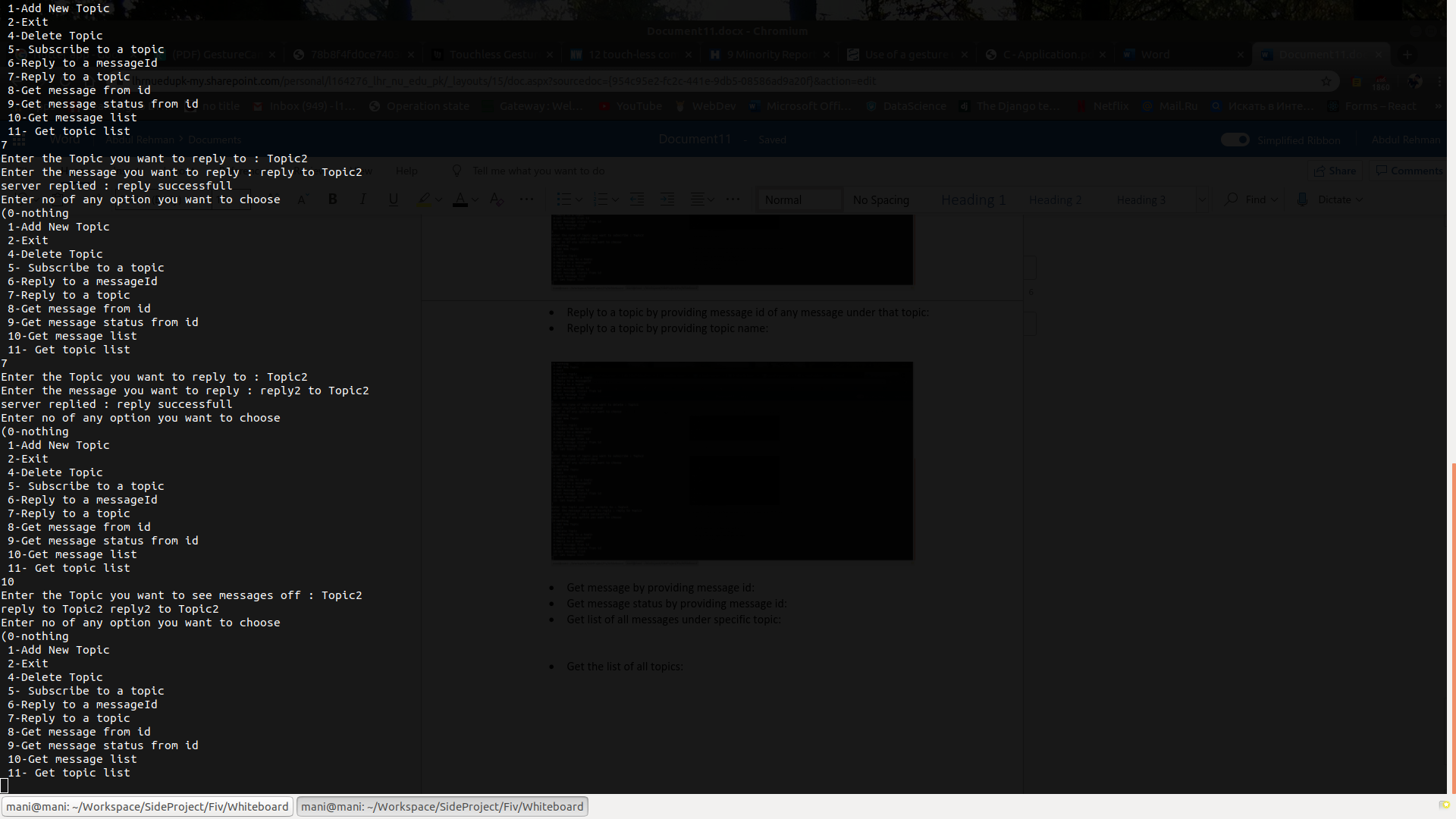
* + Subscribing to a topic:



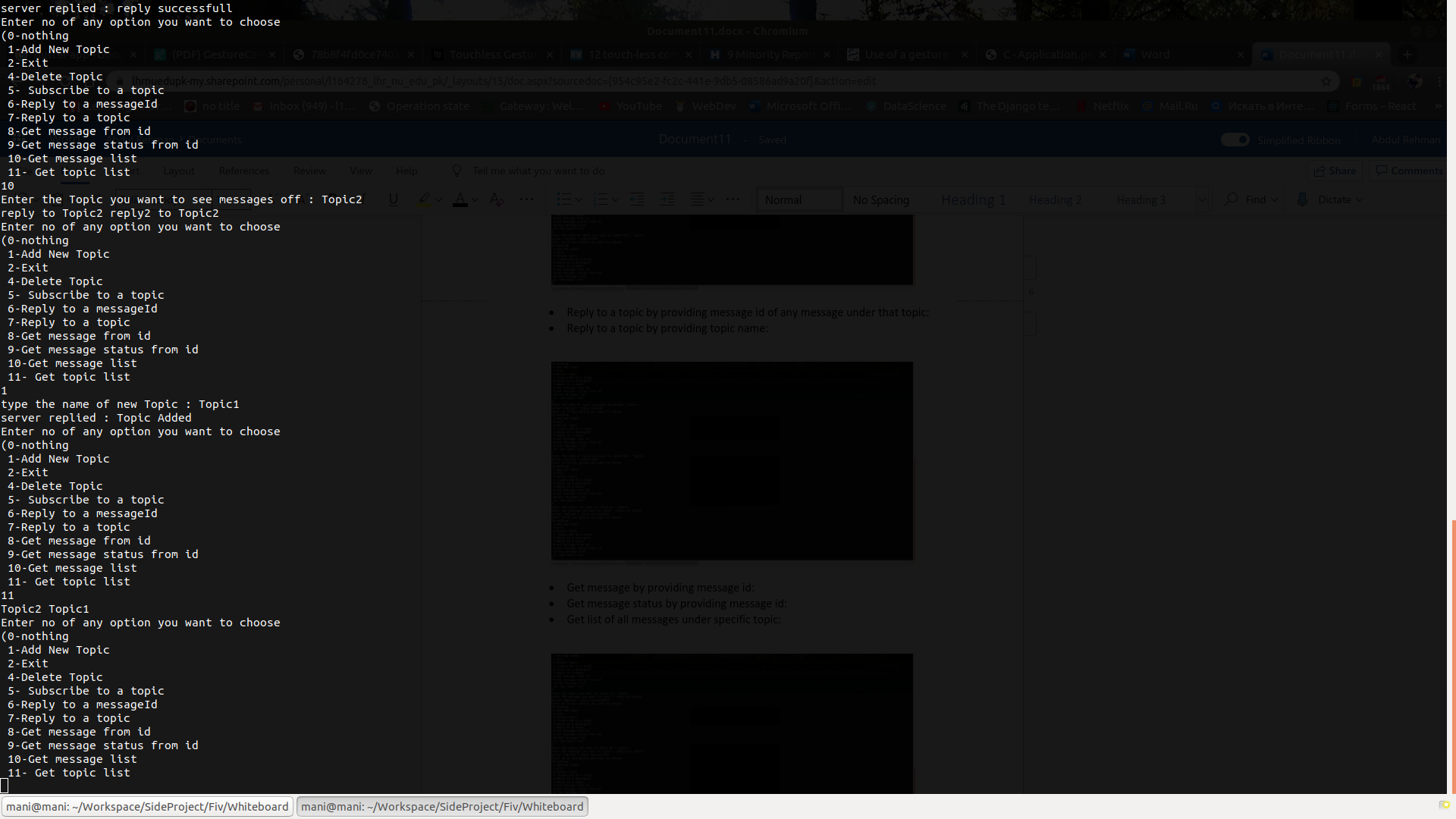
* + Reply to a topic by providing topic name:



* + Get list of all messages under specific topic:



* + Get the list of all topics:



**Release Notes:**

This program was intended to provide each functionality as described in the document provided. Although it has some limitations.

To run the program first ICP\_Server.c file has to be compiled and then its object file is run. For clarification typing these two commands in order while in project directory:

* + gcc IPC\_Server.c -o Server
  + ./Server

These commands will start the server side. Server should be in running state in order for client to interact with it. To run client side of this project following commands are to be typed in separate terminal while in project directory:

* + gcc Client.c -o Client
  + ./Client

Explicit error handling and dealing with misbehaved clients is still under development, but the code doesn’t crash while performing tasks in correct order and with correct commands as also described in testing version. Errors are only expected if there is a problem with SYS V semaphores.