Documentation & Engineering Notebook

0 User Interface

See `README.md` for how to run the server and client programs.

User interface is the same for both the socket implementation and the gRPC implementation. The client's program provides the following functions for the user:

Function	**Brief description**
(1) Create [username]	Create a new account with the given username
(2) Login [username]	Log in the account with the given username
(3) List [name]	List the accounts with matching names
(4) Send [target_user] [message]	Send [message] to the account [target_user]
(5) Delete	Delete the current account
(6) Logout	Log out
(7) Fetch	Fetch all unseen messages
(8) Exit	Exit program

The user inputs the number of the function and will be prompted to input the parameters for the function. Then, the client program will send requests to the server to perform the corresponding function. Results will be displayed.

Error messages will be displayed if the user inputs an invalid function number or invalid parameters.

The client exists if it cannot connect to the server (due to server crash, for example).

Some requirements:

- A username contains at most 20 characters of upper or lowercase alphabetical letters or numbers.
- [message] cannot exceed 500 characters.

Error messages will be displayed when these requirements are violated.

1 High Level Design

In this section, we describe our high level design. This design is the same for both the socket implementation and gRPC implementation. Then in Section 2 and 3, we will describe the two implementations respectively.

We have a single server that can be connected by multiple clients. The server is stateless except that it keeps two global data structures: a list of existing accounts/users and a database that records all the messages sent by one user to another. For each user, the server maintains an ID for each message sent to that user; ID starts from 1 and increases continuously. A lock is used to prevent multiple server threads from modifying global data simultaneously.

The client and the server interact in a request-response manner. The server provides the following services to the client:

1. **Create_account [username]:**
   Create an account with username [username]. This account is added to the server's list of existing accounts.
2. **Check_account [username]:**
   Check whether the account [username] exists (in the server's list of existing accounts).
3. **List_account [wildcard]:**
   List all the accounts whose usernames match with the pattern [wildcard].
4. **Delete_account [username]:**
   Delete the account with [username]. This account is removed from the server's list of existing accounts. All the messages sent to this account are deleted from the server's database (including undelivered messages).
5. **Send_message [username] [target_name] [message]:**
   Send a message [message] from the user [username] to the target user [target_name]. The message will be added to the server's database.
6. **Fetch_message [msg_id] [username]:**
   The client asks for all the messages sent to the user [username], starting from the [msg_id]-th message. The server returns these messages to the client.

An important design decision we made is that the server does not provide **login** and **logout** services to the client. These two functions are implemented *by the client*, in the following way:

- To login with a given [username], the client asks the server whether [username] is an existing account (via the **check_account** request). If yes, the client records this [username] in a local variable, regards itself as "logged in", and then starts to fetch all the messages sent to this user (via the **fetch_message** request).
- To logout, the client clears the local variable [username], without communicating with the server.

The main reason why we made this decision is because we think that there is no need for the server to know whether a user has "logged in" or not, at least in this assignment. In reality, login is used as a mechanism to associate an account with a particular device, so as to prevent other devices from logging in the same account and making critical requests (like sending messages).

This requires some authentication techniques that are beyond the scope of this assignment. One might think of implementing login by associating an account with the socket connection between the client and the server. However, we argue that this design is bad: (1) A socket is usually used for only one message exchange[1], but associating a logged-in account with a socket requires the socket to be alive for a long time. If the socket is broken (due to, e.g, program crashes or network failures), the user has to log in again. This is undesirable. (2) If the socket is closed on the client side but not closed on the server side, then the client cannot log in.[2] (3) In the gRPC implementation, the low-level sockets are hidden, so we cannot define "login" using sockets. Therefore, given that authentication is not needed in this assignment, we think that the server does not need to record whether an account has logged in.

In the **fetch_message** request, we require the client to provide [msg_id] to specify which set of messages to fetch, instead of letting the server record which messages the client has fetched and has not fetched. This design keeps the server stateless and prevents inconsistencies between the server and the client (for example, the server sends a message requested by the client but the client does not receive the message due to network failures).

2 Socket Implementation (Wire Protocol)

The files corresponding to the socket implementation are `server.py`, `client.py`, `serverFunction.py`, `clientFunction.py`, and `protocol.py` in the root folder. The unit test is in `unit_test/test_client.py`, which is a client that sends various requests to test the functions of the server.

The server listens for incoming socket connections. To request a service, the client creates a socket connection with the server and sends a request message. After performing the service, the server sends a response message (or a sequence of responses) to the client, and then closes the socket. We do not use a socket to handle multiple requests. This is consistent with the convention that "client sockets are normally only used for one exchange".

2.1 Client's Request

The request message sent by the client is a binary string in the following format (this format is the same for all of the 6 services above):

```
request_message = message_len + message_body
```

where `message_len` is a 2-byte unsigned int representing the length (number of bytes) of the message, encoded using the network byte order (big-endian). We require the message be at most 20000-byte-long, so a 2-byte `message_len` is enough. The message body has 5 fields:

```
message_body = op + status + username + target_name + message
```

[1] https://docs.python.org/3/howto/sockets.html

[2] This happens when, e.g., the client's network gets disconnected first, then the client program crashes and restarts. The client now cannot connect to the original socket. The server's socket will wait for receiving messages for a long time before timeout.

The `op` (operation code) is a 2-byte signed int indicating which service/operation the client is requesting, encoded using the network byte order. The following 4 fields (`status`, `username`, `target_name`, `message`) represent parameters given to the server to perform the service: `status` is a 4-byte signed int, encoded using the network byte order; `username`, `target_name`, `message` represent strings, encoding by concatenating the length (number of bytes) of the utf-8 encoding of the string and the utf-8 encoding itself. For example, if the `username` field represents the string "user1", then it is

`username = len(utf-8("user1")) + utf-8("user1")`

The lengths are used by the server to parse the binary string, in particular, to separate the 3 fields `username`, `target_name`, `message`. The "+" symbols are for illustration only; they are not in the actual message.

The following table shows the operation code of each of the 6 services and how those services interpret the 4 fields `status`, `username`, `target_name`, `message` as the parameters. An empty slot means that that field is not used in that service. The client can put any valid value (or the default value) in that field; it will be ignored by the server.

Service	`op`	`status`	`username`	`target_name`	`message`
create_account	1		[username]		
check_account	2		[username]		
list_account	3				[wildcard]
delete_account	4		[username]		
send_message	5		[username]	[target_name]	[message]
fetch_message	6	[msg_id]	[username]		

2.2 Server's Response

Upon receiving a request message from the client, the server parses the request message, chooses the service according to the operation code `op`, and obtains the needed parameters from the request message according to the above table. After performing the service, the server sends back to the client a response message. For convenience, we let the response message follow the same format as the request message (but with different interpretation):

`response_message = message_len + message_body`

where

`message_body = op + status + username + target_name + message`

The operation code `op` is the same as the operation code of the request message. The `status` (2-byte signed int) is a status code indicating the result of the operation. For example, `status=0` means that the operation "succeeds" and a non-zero `status` might indicate an error like "account does not exist" (see `protocol.py` for details). The following 3 fields

`username`, `target_name`, `message` are text messages returned to the client. In particular, `message` is either an error message or an "actual" message that the client receives from another client (when the client fetch messages).

Except for the **list_account** and **fetch_messages** services, the server only sends one response message to the client. For **list_account** and **fetch_messages**, the server will continuously send multiple response messages. The reason why we need multiple response messages is because the list of accounts or the set of messages to return to the client can be very long, which might exceed the length limit (20000 bytes) of a single response message. (We limit the length of a request or response message for security reasons.) When sending each of the multiple response messages, the server uses the `status` field to indicate whether there are more response messages to be sent. The client is responsible for receiving all response messages until `status` indicates "No Next Element".

The following table summarizes the possible response messages the server might return after performing a service. In our implementation, no response message will use the `target_name` field (set to the default value), so this column is omitted. The last column explains the condition under which the server will return the corresponding response message; usually, this is also included in the `message` field for the client's reference.

Service & Parameters	`op`	`status`	`username`	`message` (the reason for this response)
create_account [username]	1	Success		
		Invalid Username		[username] is invalid
		General Error		User [username] already exists
check_account [username]	2	Success		User [username] exists
		Account Not Exist		User [username] does not exists
list_account [wildcard]	3	No Element		No account is matched with [wildcard]
		Next Element Exist	A username that matches with [wildcard]	(There are more such usernames; the server will continue sending responses.)
		No Next Element	A username that matches with [wildcard]	(All such usernames have been sent; the server will stop sending responses.)
delete_account	4	Success		

[username]		Account Not Exist		User [username] does not exist
send_message [username] [target_name] [message]	5	Success		
		Account Not Exist		The sending user [username] does not exist
		Account Not Exist		The receiving user [target_name] does not exist
		Message Too Long		The [message] is longer than 500 characters
fetch_message [msg_id] [username]	6	Account Not Exist		User [username] does not exist
		General Error		[msg_id] < 1
		No Element		There is no message to fetch: namely, [msg_id] > the total number of messages sent to user [username].
		Next Element Exist	The sender of the message	`message =` a message (There are more messages to fetch; the server will continue sending responses.)
		No Next Element	The sender of the message	`message =` a message (There is no message to fetch; the server will stop sending responses.)

2.3 Other Implementation Details

To facilitate the sending and receiving of messages over socket, we implement a message class `Message` (in `protocol.py`) that includes methods `encode()`, `decode_from(binary)`, `send_to_socket(s)`, `receive_from_socket(s)` that can encode the message object into a binary string, decode a object from a binary string, send the object via socket (after encoding), and receives an object from socket (using the decoding function). This design is inspired by the Protocol Buffer. This message class is used for both request and response and for all services.

3 gRPC Implementation

The files corresponding to the gRPC implementation are in the `gRPC` folder. The unit test is in `unit_test/test_client_gprc.py`, which is a client that sends various requests to test the functions of the server.

Six essential services to ChatRoom are defined in `service.proto`.

- `rpc rpc_create_account(User) returns (GeneralResponse){}`
- `rpc rpc_check_account(User) returns (GeneralResponse){}`
- `rpc rpc_list_account(Wildcard) returns (stream User){}`
- `rpc rpc_delete_account(User) returns (GeneralResponse){}`
- `rpc rpc_send_message(ChatMessage) returns (GeneralResponse){}`
- `rpc rpc_fetch_message(FetchRequest) returns (stream ChatMessage){}`

These correspond to the six essential services that require client-server communication. The other peripheral functions like `login` are supported through one or multiple of these services (see the “High Level Design” section). Note, just like in the socket design, `rpc_list_account` and `rpc_fetch_message` takes in a single request object from the client and returns a stream of responses. Correspondingly, the following message classes are defined. Among them, class `ChatMessage` has a `status` field to indicate if there are no chat messages sent to `username`, no further messages to be fetched, or there are errors; `Wildcard` is a string input class.

```
message User {
    string username = 1;
}
message ChatMessage{
    string username = 1;
    string target_name = 2;
    string message = 3;
    int32 status = 4;
}
message Wildcard {
    string wildcard = 1;
}
message FetchRequest {
    int32 msg_id = 1;
    string username = 2;
}
message GeneralResponse {
    int32 status = 1;
    string message = 2;
```

```
}
```

The client stub and server skeleton are generated in `service_pb2.py` and `service_pb2_grpc.py`. The client maintains a stub through the gRPC channel, and calls one of the services above to get a `response` object that records the result of the service (including error message, if any). The server maintains a skeleton through the gRPC channel, and listens for the service requested. Passed along with the service request is a `request` object that provides the necessary information to perform the task. The server then returns a `response` object to the client. The overall logic is similar to the socket implementation.

4 Comparisons: Socket vs gRPC

4.1 Buffer Sizes

The following table shows the lengths of the buffers (i.e., the number of bytes of the encoded message) sent back and forth between the client and the server, in a sequence of operations, in the socket implementation and the gRPC implementation. To measure the length: In socket, we implement a method `byte_length()` in the message class `Message` (in `protocol.py`); in gRPC, we use the method `msg_obj.ByteSize()`. The server outputs the lengths of the request and response messages to screen.

Operation	**Socket buffer size**		**gRPC buffer size**	
	Request	Reponse	Request	Reponse
Create “user1”	19	67	7	55
Create “user2”	19	67	7	55
Login “user1” (including check account and fetch messages)	19 (check) + 19 (fetch)	29 (check) + 55 (fetch)	7 (check) + 9 (fetch)	17 (check) + 45 (fetch)
List account “*”	15	19 (user1) + 19 (user2)	3	7 (user1) + 7 (user2)
Send message (from “user1”) to “user2” “this is a message”	41	39	33	27

Send message (from “user1”) to “user2” “I love this class!”	42	39	34	27
Logout (“user1”)	N/A	N/A	N/A	N/A
Login “user2” (including check account and fetch messages)	19 (check) + 19 (fetch)	29 (check) + 36 (msg1) + 37 (msg2)	7 (check) + 9 (fetch)	17 (check) + 28 (msg1) + 29 (msg2)
Delete (“user2”)	19	76	7	64
Logout (“user2”)	N/A	N/A	N/A	N/A

We see that the socket buffers are slightly larger than the gRPC buffers. This is roughly because we use the same encoding format for all messages in our socket protocol, which includes some redundant parts for some messages.

4.2 Complexity of the Code

gRPC implementation is definitely shorter and more convenient than defining our own wire protocol and the way of encoding. For gRPC, the server side only needs to implement the service functions required, by receiving the request objects and returning the response objects. This is much easier and cleaner than the wire protocol where the server has to worry about sockets, catching all the failures, and sending. The client side is also easier as the client stub handles all the blocking send requests and get responses methods. So way cleaner!

4.3 Performance Difference

We saw no visible difference between our own implemented wire protocol and the gRPC implementation.