Design Document

This is a mini Social Networking Service where multiple users can Login and communicate. After successfully logging in, they can Follow/Unfollow fellow users (if they exist!), List all the users and the users that follow the current user, and finally get into Timeline. In Timeline, users can post text, and other users that follow the current user will get the updates on the go/live. However, a user can see the posts from the moment or follow and the recent 20 only.

Let's go step wise and check how this works.

Initial login

Approach: It basically checks if user is present in client_db at the server, given the username.

As soon as a user starts client code, a stub is generated with the provided hostname and port. The connectTo() function does that.

The stub allows us to invoke a method on the stub object, passing the necessary parameters. The method call appears as if we are invoking a local function.

Now Login() function is called and here a gRPC request is sent using
stub->_Login(&context, request, &reply) to Login(ServerContext *context, const Request *request, Reply *reply) on the server side.

In the server side, it checks if user already exists in client_db.
If it doesn't exist, a new user is created and the user name is set.
If user already is present, "reply" object's msg parameter is set with set_msg()
with the applicable error code(FAILURE_NOT_EXISTS)
If above condition doesn't fail, it creates a timeline file for the user so that the messages are persisted.

Now, we are able to run commands.

For every line of input, processCommand() is called, and the commands are parsed. If arguments are given, those are parsed as well, and the particular function is called.

Follow

Approach: Given the current username, and following username, I added each other objects into the vectors respectively.

The function definition, in the client point of view has the sending object as Request, and receiving object as Reply.

Request object has a string username and repeated string 'arguments' as arguments. We can think that repeated string is something like an array that protobuf provides, hence we have to use it's APIs for access and setting.

On the client side, Follow() is executed.
Over here, username is filled with current user's name, and an argument is appended to 'arguments' array with the username. This name is username that the current user wants to follow.

After calling stub_->Follow(&contex, request, &reply);
The counter part in server with the same function name is called.
Over here, it searches for the current user, if it doesn't exist throw error.
If found, it searches for the following username, if it doesn't exist throw error.

Now, since both current user and following user exist,

1. I add following user to current user's client_following vector.
2. Similarly I add current user to following user's client_follower's vector.

I make sure to check beforehand if the user already has been added to the vectors and that it's not replicated.

Once everything is fine and good Status::OK is returned from the server, **without** any error message being set.

How am I handling errors?

I am not using gRPC errors as it is reserved for gRPC related errors by conventions. Hence all the application related error's are set as reply message and return status is Status::OK.

I kept the error identifiers as the ones already present in skeleton code. The error strings returned from gRPC are parsed for the identifier and IReply's comm_status is kept the same. To help set comm_status, I've defined isErrorCodeExists(reply.msg(), "IDENTIFIER") function to check if IDENTIFIER exists in the reply message. Examples of identifiers: FAILURE_ALREADY_EXISTS, FAILURE_NOT_A_FOLLOWER, etc.

Unfollow

This is similar to Follow function. If a user is present in current user's following list, I remove the user. Similarly I remove current user from userToUnfollow's follower's list.

List

The above three functions used Request, Reply objects defined in sns.proto.
List function requires little different definition than the provided Request and Reply. Since we need to return all the users present in the system along with the users that are following the current user, we need a different data structure, something like an array.

As mentioned before, protobuf provides 'repeated' keyword to define an array-like object, i.e a field to hold multiple values of the same data type. Hence we are using ListReply which contains 2 repeated string fields.

Approach:
When the client calls LIST, a gRPC call carrying the username is executed. At the server side, the ListReply object is populated:

```cpp

// client_db is looped to get all users
for (int i = 0; i < client_db.size(); i++)
    {
    //we need to use add_{argument name} on the repeated field for appending
      list_reply->add_all_users(client_db[i]->username);
    }

// client_followers is looped to get all the followers of the current user
    for (int i = 0; i < curUser->client_followers->size(); i++)
            {
      list_reply->add_followers(curUser->client_followers->at(i)->username);
    }
```

Once server side execution is done, fields 'all_users' and 'followers' are iterated over to fill the IReply object. The repeated fields have cpp-like iterators, and hence i used them.

```
for (const std::string &user : list_reply.all_users())
    {
      ire.all_users.push_back(user);
    }
```

Timeline

Client

To use streaming in gRPC, we need to initialize a different stub, rather than the one we use for unary communication. That new stub is the streaming object.

```
 std::shared_ptr<ClientReaderWriter<Message, Message>> stream(
      stub_->Timeline(&context));
```

stub_->Timeline(&context) tells gRPC to call Timeline function in server. And associate the current client stream with the stream argument in the function definition at the server.

Since this is a bidirectional stream, read-write and write-read operations are completely independent. Client can stream to write while Server can read, and vice versa.

I take advantage of that, and create a cpp thread, just so that the client can *continuously* stream write messages, while the other main thread can *continuously* stream read.
I am in write thread forever by while(1) loop, and in read loop by while(stream->Read(&server_msg)) . As long as stream is not ended on server side, the client
stream->Read would be returning true.

A small change I did was add `int32 is_initial = 4;` to Message definition in sns.proto. This would be set to 1 , if the message sent is the initial message, else it would be defaulted to 0.
The client side sends a dummy message with is_initial set to 1 as soon as user enters Timeline mode.

Server

In the Timeline function we are provided with

```
ServerReaderWriter<Message, Message> *stream
```

pointer, which we can use to read and write messages in the server.

As with the client, I *continuously* read from the client.
In the dummy message that comes from the client,

1. The server saves the stream object of the user.
2. Reads the last 20 messages from the user_tl.txt file. Streams it back to the user.

The user_tl.txt file is the storage of all the posts that the user received. The top of the file always has the latest post.

Now, for every message which comes from a user that has posted, the server reads, then

1. It appends the message to the timelines of the users, that are following the **current** user. We find who are they by the client_followers vector. Function used is "append_to_timeline(follower name, current name, current timestamp, current message)"
2. It streams the message (timestamp,username,content) to the followers. As we have saved the stream object whenever a client is in timeline mode, the program uses that stream object (if it exists, implying user in timeline mode, else ignore), to write the message to all the followers.

Now we have achieved Timeline feature.

Resources Used

1. https://protobuf.dev/reference/cpp/cpp-generated/
2. https://grpc.io/docs/languages/cpp/basics/