Trevor Ledbetter, 326001657
Anthony Nguyen, 826005614
CSCE 438 Section 500

Project 2

Compilation Instructions

Our project can be compiled with the provided makefile, which will create the client executable **`fbc`** and server executable **`fbsd`**. First execute "**`./fbsd <port number>`**", followed by "**`./fbc <hostname> <port number> <username>`**" in a separate terminal or connected VM. If the server prints an error, try a different port. We also assume our code will be run on a POSIX system like the one provided in the VM, several file operations rely on this.

Design

This project was implemented using gRPC and Protocol Buffers 3. The design plan is organized by RPC and Protocol Buffer design, server implementation, and client implementation. The end goal is to create a tiny SNS implementation that can handle many connected users and keep all interactions in persistent memory so that they can be restored in the event of a server failure.

Remote Procedure Calls (RPCs)

The project consists of a single Service, **`SNS`**, which has the following RPCs:

1. **`InitConnect (ClientConnect) returns (ServerAllow)`**
    - RPC which is called upon client construction. Notifies the server that a user is connecting to initiate necessary setup.
2. **`Disconnect (ClientConnect) returns (ServerAllow)`**
    - RPC which is called before the client exits. Notifies the server that the user is disconnecting to initiate necessary cleanup.
3. **`Follow (FollowRequest) returns (FollowReply)`**
    - RPC which is called when a client wishes to follow another user.
4. **`Unfollow (UnfollowRequest) returns (UnfollowReply)`**
    - RPC which is called when a client wishes to unfollow another user.
5. **`List (ListRequest) returns (ListReply)`**
    - RPC which is called when a client wishes to receive a list of all current users in the server database.
6. **`Update (UpdateRequest) returns (UpdateReply)`**
    - RPC called at regular intervals from the client that receives any relevant and new data from the server.

7. **`SendPost (Post) returns (PostReply)`**
    - RPC called when the client posts a message in timeline mode
8. **`ExecDebug (DebugRequest) returns (DebugReply)`**
    - RPC called for debugging purposes during development.

Protocol Buffers

The project consists of the following protocol buffer definitions:

1. **`ClientConnect`**
    - Message sent in **`InitConnect`** RPC that contains details about the client that wishes to connect to the server.
2. **`ServerAllow`**
    - Message received in **`InitConnect`** RPC that contains details about the client connection to the server.
3. **`FollowRequest`**
    - Message sent in **`Follow`** RPC that contains details about the client and the user the client wishes to follow.
4. **`FollowReply`**
    - Message received in **`Follow`** RPC that contains details about the client follow request from the server.
5. **`UnfollowRequest`**
    - Message sent in **`Unfollow`** RPC that contains details about the client and the user the client wishes to unfollow.
6. **`UnfollowReply`**
    - Message received in **`Unfollow`** RPC that contains details about the client unfollow request from the server.
7. **`ListRequest`**
    - Message sent in **`List`** RPC that contains details about the client requesting a list.
8. **`ListReply`**
    - Message received in **`List`** RPC that contains the requested list information.
9. **`UpdateRequest`**
    - Message sent in **`Update`** RPC that contains details about the client requesting fresh data from the server.
10. **`UpdateReply`**
    - Message received in **`Update`** RPC that contains fresh data for the client if there is any.
11. **`PostReply`**
    - Message received in **`SendPost`** RPC that contains information relevant to how a post was handled by the server.
12. **`Post`**
    - Message sent in **`SendPost`** RPC that contains information relevant to a user's timeline post.

13. **`Timeline`**
    - Message containing information relevant to a user's timeline for server serialization purposes.
14. **`User`**
    - Message containing information relevant to a user for server serialization purposes.
15. **`UserDataBase`**
    - Message containing information relevant to the in-memory database on the server for serialization purposes.
16. **`DebugRequest`**
    - Message sent during an **`ExecDebug`** RPC call for testing purposes.
17. **`DebugReply`**
    - Message received during an **`ExecDebug`** RPC call for testing purposes.

Server Implementation

The server design is a synchronous design which handles each RPC call individually and performs the necessary logic in sequence. The general logic flow is as follows: receive an RPC call from the client, execute the logic necessary, modify the database in memory and in persistent memory as needed, send a response to the client. The server has the following members and methods:

- SNSImpl
    - Contains the gRPC implementation of the SNS service.
    - Holds database of all users
    - Handles RPC calls
- Post
    - Data structure which represents the information pertinent to posts on a user's timeline
    - Contains poster name, timestamp, and message
- User
    - Data structure which represents the information pertinent to a user.
    - Contains username, timeline, followers, users following, whether they are connected through a client, as well as other server runtime variables needed to ensure data is synchronized with the client when an update is requested.
- RPC Implementations
    - All the listed RPCs are implemented to satisfy each client's request.
    - They are implemented in a synchronous Service
    - All RPCs which modify the database also save to persistent storage to ensure data persistence in the event of a spontaneous crash.
    - All logic is initiated by the client, the server only responds with the appropriate data for each call and ensures all clients requesting an update are synchronized with their followers and timelines.
- Program Execution (**`main()`**)

    - The program can be called as detailed above and uses the port number argument when instantiating the SNS service and assigning a listening port.
    - The **`main()`** function calls **`RunServer()`** which handles the actual setup of a gRPC service

Client Implementation

The client program implements the IClient class defined in the provided client header file, and performs all necessary gRPC operations to communicate with the server. The client's members and functions are defined in more detail below.

- Program Execution - **main()**
    - Receives a port number, hostname, and user name from the user.
    - Creates a Client object and calls the run_client() function defined in the client's header, which in turn calls run().
- Connection with the Server - **connectTo()**
    - Called by run(), this function uses the provided username and port number to create a gRPC connection with the server by creating a gRPC channel and a new stub. The client keeps track of this stub and it is used for all communication with the server.
    - Sends a connection request to the server, so the server can add this client as a user.
- Command Mode Logic - **processCommand()**
    - Parses the user's command and sends the necessary request to the server using the predefined RPCs and the client's stub.
    - Receives a response from the server and returns the nature of the response in an IReply container to run().
- Chat Mode Logic - **processTimeline()**
    - Retrieves the initial posts in the timeline from the server by calling **checkForUpdate()**, which gets the most recent posts from users that the client is following or the client's own posts.
    - Spawns a seperate thread that will check for an update to the client's timeline every 500 milliseconds, by calling **checkForUpdate()**.
    - After the update thread is created, processTimeline() loops indefinitely receiving the client's posts and sending them to the server using **sendPost()**.
    - Once in timeline mode, the only wait to exit is with CTRL-C.