Ethan Veselka

MP 2

3/7/23

Design Document:

See Attached PDF for Diagram

In summary: The TSNS design follows a Multi-Threaded 'aynchronous' approach

The general structure of the program is as follows:

- Clients connect to server using GRPC protocols, server accepts connections, and the client can make server function calls using its GRPC stub.
- Client enters a command through the interface, and then that is preprocessed before being sent to the server as a GRPC of the corresponding type.
- The server handles all commands using two main data structures: First, a vector containing structs with information about every user. Second, a .txt file system to save the current state of the server, or any client, so that the most recent state of any client who disconnects, or if the server goes offline, is reinstated. This allows for full persistence of the server and client states individually, as well as full update capabilities for client events (follow/unfollow/timeline updates) while they are offline.
- All commands return as expected and change the state of the server, with proper and reasonable error checking on commands with corresponding replies from the server. Timeline more specifically, has special behavior:
  - **Login:** Used to verify the connection of clients, creating a new instance of the user struct, or repopulating one with most recent client state if the client is logging in again after disconnecting.
  - **List:** Lists all users that exist on the server, and all users currently following the requesting user.
  - **Follow:** Follows a user, allowing the requesting user's timeline to be updated with all posts made by the followed user, until they unfollow.
  - **Unfollow:** Unfollows a user, the requesting user will no longer receive updates to their timeline from this user.
  - **Timeline:** Timeline allows the user to enter timeline mode, where they can make posts, and see posts from users they follow. There are two main functionalities, both of which are managed by threads. First, a thread will continually read from the bidirectional stream, store the messages in local timeline files, and then update the new_posts vector of all following users. Second, a thread will simultaneously first: write any updates from the current client's timeline to them, up to the most recent 20, in reverse order,

second: read posts from their new_posts vector to get real time updates from other online clients in timeline mode.

All events are managed by threads, however, there is a limit to the asynchronous ability of the server’s bidirectional streams responsible for reading and writing messages, so although all events seem to happen simultaneously, they do bottleneck at the stream, so if there were a sufficiently large amount of connected clients, the server would run much slower because communication is not *completely* asynchronous. Because of this, I have dubbed it an ‘asynchronous’ Multi-threaded approach as opposed to a synchronous one.