Comp3331 s1 2014 Assignment 1

Matthew Moss <mdm@cse.unsw.edu.au>

March 23, 2014

1 Design and Implementation

To simplify the implementation of this assignment, I used Java's built in serialization. In a real world system this is often not an option, but for a one-off assignment such as this it makes development faster and easier. The server component begins by opening a single listening TCP port. All messages to the server are sent over this port, and different handler functions are invoked by matching the class of the serialized object. Other than the list of post clients and posts, the server maintains no state.

Readers use several threads, to allow concurrent operations to happen smoothly. Blocking queues are used to simplify inter-thread communication. All user input via stdin happens via a StdinListener thread, which filters input based on content. If the input contains "y" or "n" alone, it is considered a confirmation and placed in the confirmations queue. Otherwise, it is placed in the input queue. This allows the reader to immediately see the results of the user responding "y" or "n" to a chat request. When a client starts it also opens a random UDP port and a listener thread, to handle chat.

Readers in "pull" mode are the simplest of the two types. When updated post information is needed, they send the current book and page to the server, which responds with the posts from that book and page. At that time a timer is started to run a function that queries the server again. This is repeated indefinitely at the specified interval. If a different page is displayed, the previous timer is cancelled and a new one started.

Readers in "push" mode open a TCP socket on a random port, and advise the server of this in their initial handshake. The server will then forward new posts to this port when they are created. A PostListener thread listens on the port, and prints messages when new posts are received.

Chat is accomplished with the ChatListener and ChatControlListener threads. Each reader has a list of conversations. If a reader requests chat with another reader, a ChatRequest is added to the list. This ChatRequest signifies the user is ready to accept communication from the other user. When a user receives a chat request, they look for a matching ChatRequest already in the list. If it exists, it is replaced with the newer one. This means that when a user accepts chat, they can reply with the same message format, simplifying communications.

1.1 Message Format

Messages are simply serialized Java classes. To query the posts of a particular page, the reader sends a PostList containing a book and page. The server responds with a PostList containing a book, page, and list of posts.

To register for push, a reader sends a Client message. The server adds this to its internal list of push clients. The client message contains the host and port of the client, and the list of known posts. The server checks these known posts against its database, adds any unknown posts to a PostList and replies with this list to the client. Thus the initial handshake is completed in a single request/response.

The Post message is used to create new posts. It contains an id, the relevant user name, and the book, page and line of the post. The id field is ignored by the server, as it is automatically assigned when the post is inserted in the database. As allowed in the spec, the server makes no response to a Post message.

The ChatRequest message contains fields for the user sending, and user receiving, as well as the host and port of the sender's chat socket. With this message, a user is able to open a UDP connection to chat to a client.

The chat messages contain the sender's user name, followed by a colon (":") and the chat message. This means parsing the string to check if messages are allowed is easy, and the string can be printed in full to display the chat message.

1.2 Trade offs

- This is a very insecure system. String matching on user names is used for all user identification. This is the first of a long list of potential issues that I won't expound here.
- Some operations could be optimized, but performance is not a target of this design.
- If performed incorrectly, many operations will cause unpredictable errors. For example, if the display command is passed invalid arguments, the file reader will throw an uncaught exception. In my opinion this is acceptable, as friendly user interfaces are not a requirement of the assignment.
- The chat string (user name + ":" + message) has a maximum length of 256. Longer messages can be sent, but will not be displayed.
- If the user name contains a ":" symbol, chat messages will not be handled correctly. Effectively, every message will be from an unknown sender, and will be ignored.

1.3 Possible Improvements and Extensions

A persistent database of users would make a nice addition to the server for this assignment. Allowing users to maintain lists of "friends" from whom they automatically accept chat could improve the user experience.

2 Attributions

To the best of my knowledge, all submitted code is my own.