CSC 536: Distributed Systems II
Professor Ljubomir Perkovic
Final Project Report
By Kevin Westropp
06/11/2014

2. A real-time social networking application "Chat Share Web App"

1.) How to Run My Project

Web Server

Navigate to ChatShare/chatshare directory inside of zip folder.

Compile with → $ sbt compile

Then run web server with → $ sbt run

Client

Open web browser and navigate to http://localhost:8888/

You will prompted to login. Please enter a username to use the web app.

To Test

Recommended to open multiple web pages and navigate each to http://localhost:8888/

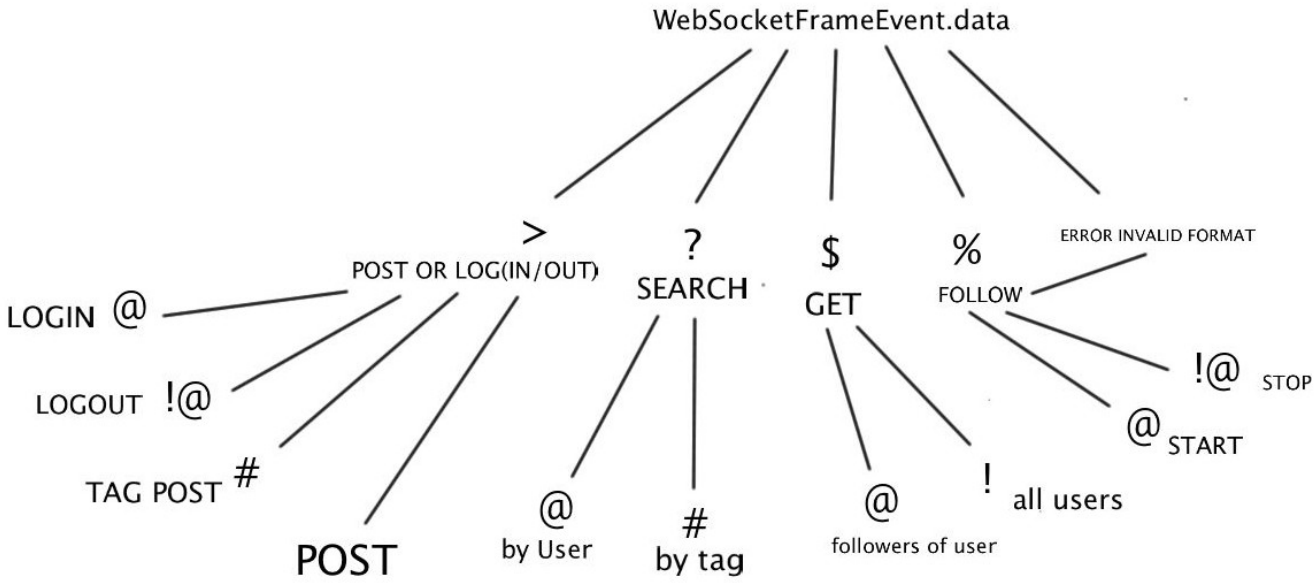
Login with different usernames. Then post messages with #tags and click "Update" under Users to see other users of the system. You can click on a user under "Users" to start following them. You can also search messages by other users with @username or search messages by tags with #tag.

2.) Overall System Design

The overall system starts at the web server (chatapp.scala) which uses the socko web server framework. It defines the http request which always defaults to serving back the index.html which is the only html file needed to run the application. It then creates a web socket on the client side (web browser) which passes data back and forth between the server and the client. The index.html uses javascript to handle all of the user interaction and passing data between the user input and web server. The index also makes use of Twitter Bootstrap which is a front-end design framework that makes front-end design much easier.

Whenever the ChatApp Object (Singleton) receives a web socket frame event from the client it then creates and forwards this to the ChatHandler actor (chathandler.scala). The ChatHandler actor then reads the event data to determine what type of message this belongs to and either forwards this request to the Router actor (routersupervisor.scala) or directly responds to the client.

An outline of the tree of message types is below:



Depending on the message content and prefix hierarchy, the message then goes to the router as whatever event type this is corresponding to (post, search, get, follow).

The router actor (singleton) is created at the web server (chatapp) and a reference is sent to the ChatHandler with each request. Then once the router gets the a message from the ChatHandler about what type of message it is, it then forwards this on to either the ClientHandler actor (Singleton), UserHandler actor (Singleton), or the MessageHandler actor (Singleton). The router creates each handler and supervises each handler and routes messages between each as well as the ChatHandlers.

The ClientHandler is in charge of the clients and keeps a map of the socket ids by username. It also handles connecting and disconnecting as well as forwarding certain requests to the UserHandler, such as logging in and also requests to follow certain users.

The UserHandler is in charge of the users and keeps a map of the usernames and list of users that this user is following. It also has a reverse map of the users and a list of who is following them. This makes it much quicker to respond to both types of inquiries but also requires more work keeping everything up to date. It also steps up lists when a new user connects to the system.

The MessageHandler is in charge of the messages and keeps a map of the messages by username and a map of the messages by tag. It is responsible for storing and providing back requests for all the messages sent in the system.

3.) Benefits to Design

The lifecycle of every ChatHandler Actor is a simple receive, process and stop cycle. I think this makes processing requests much faster. I could have made the ChatHandler persist as long as the client connection remained open but then I could foresee two points of failure, one at the client and another at the ChatHandler. With this design, if a ChatHandler fails the client connection is not lost as well.

I felt the router/supervisor was necessary for supervising the handlers (client, user, messages) using the actor supervisor strategy from akka because it could potentially recover if one of these actors dies or throws an exception. I also use the ask pattern with a future in the router as well as the ClientHandler to forward processing requests to other actors. That way if there is an exception somewhere in the processing of the request the system can recover and potentially move on from an unexpected exception.

I wanted to implement a couple more things but ran out of time and still might get around to finishing these up in my free time. I wanted to find a way to easily swap out the Handler if there was a problem and was going to write a special object that would switch references but I ran out of time to implement this part. Another aspect that I was looking to implement but ran out of time, was the persistence layer. I was hoping to write out each message, username, data, etc. to a log file and read that back in every time one of the actors/handlers were started. This would be in place of a potential database solution which would be the preferred solution in this case.

4.)

Synchronization

Posts or messages are delivered after they are registered with the system and then forwarded to the followers of the user that made the post. These show up in real time because of the web socket, which is a major advantage in this system. Updates to the followers are handled whenever there is a user change which is reflected almost immediately upon the change being registered with the sytem. Updates to the users, or showing what users are active is implemented by letting the user click to update the users list. I would rather not bog down the system with a repeated request for updates to the users because it is something that is not needed in real time but rather on demand. It resembles more of a read your writes style of updates. While the followers and messages follow a strict consistency update model.

Replication

Replication occurs almost immediately with regards to messages and followers as long as your are logged in when a post happens and are following the person that made the post. You can also search for any post and this is a strict consistency. The users list is updated whenever the client makes an

update request and this follows a read your writes style consistency.

Fault Tolerance

Fault tolerance is handled by supervision (router supervizes handlers) and the web server supervising connections. And reads or writes to or from the client are almost always insured as long as the web socket is open on the client's side and the user has been registered with the web server. Obviously, there might exist an instance where something fails like the network but I feel this system would recover well because it warns the user on the client side if the socket is not open. Any message that is sent through to the server is almost guaranteed a response even if there is an exception or error on the server side.