ECS713U Individual Coursework Report

Kem Ibodi 200528247

Datatypes:

While coming up with concepts for the user and message datatypes I initially decided to model users to store their name and the number of messages they have sent and received. While I didn't change my initial approach for the data type I ended up with two solutions for how information is stored for each user (more details in the implementation segment). For the message data type I originally stored the message content, the sender, and the receiver. Later in development I decided to change this to implement my extra feature. In the current version I have fields for content, sender, target, and recipient. The sender of a message will only specify the target of the message as a string, while the recipient of the message will acknowledge the message and add the recipient as a user data type (more details in the implementation segment). The user datatype is made an instance of the type classes Eq and Show, with equality being justified if the usernames of two users are the same and the string version prints out a user's information in three rows. The message datatype is only made an instance of Show as the program does not need to check for message equality.

Implementation:

For my implementation my main function first creates 3 MVars, the first stores a single message, the second stores a list of messages and the last stores a list of users. My idea for this architecture is that a user process can send a message by putting a new message with a random target into the letterbox. All users will then check the letterbox and see if the target of the message matches their name. My extra functionality is that a message is only acknowledged by the system when a user confirms they have received it. For this to work messages are only added to the message list when they are received (not when they are sent). The main function then calls the generate-users function which recursively loops through a list of names to create the list of users as well as fork a thread for each user. Doing so it calls the user-process function 10 time in 10 separate threads. One of the major challenges I ran into was overcoming a deadlocked state where all thread would wait to approve a message but no thread sent the initial message, reversing the order of check for receiving and sending messages would not solve the problem as this would lead to only one thread being able to send a message, making all other threads wait to send their own including the target of the message making everyone wait. My solution to this was to create two sub-threads per user thread. One would listen for a message to the user by checking the letterbox MVar while the other has a 50% chance of sending a message to another user at random (users can't send messages to themselves). Each user's super-thread sleeps for a set amount of time before killing its sub-threads and sleeping again (this time for a random amount of time) before either recursively calling itself or terminating if the message-list has length 100. While the sub-threads are executing, I needed a way to make the main thread wait for them. I did this by having a function recursively check the message-list's length and only proceeding when 100 messages have been sent. To count messages sent and received by each user I initially had each user thread update its user object with every recursive call and then update the user-list MVar, with this approach however I was consistently running into an issue where after execution 100 messages would have been sent but the individual values for each user would not add up to 100. While experimenting I found that this was most likely a timing issue, however a simpler solution was to centralise the counting of messages per user into the main method. While this added an additional n^2 of time complexity to the program it wasn't an unreasonable change and led to full consistency. Finally, the data is printed to the terminal and the program terminates.