# **ECS713U/ECS713P - Functional Programming - Matthew Kingsbury - Report**

This report summarizes my implementation for the individual project. The program simulates a social network of ten users sending each other messages at random intervals, correctly implementing concurrency. As an extension, I implemented a ‘friends’ feature to the program, where two users are considered friends if they have sent each other a message. For each user, the program then outputs the user’s name, the number of messages they received, and their list of friends.

I created a Message data type which contains the ‘text’ of each message and which user sent that message to the current user as the ‘from’ field, both as Strings. Storing the name of the sender was crucial for implementing the friends feature of the program, so we can track which users had sent each other messages. The User data type stores the user’s ‘name’ as a String, as well as an MVar ‘messages’ as a list of Message, which stores the list of messages that user has received. As the messages list can be updated by multiple threads concurrently, storing the list as an MVar is critical for thread safety, so only one thread can update the list at any given time.

Initially I faced difficulties understanding how to know when to terminate the threads loops, as I wasn’t sure how to track the total number of messages sent. I then realized this is similar to the ‘coin flip’ example - each thread can increment a ‘numMessages’ MVar every time they send a message, and if that MVar reaches 100 then put True into another initially-empty MVar ‘check’ which the main process is trying to take. Therefore, the thread processes will end after the 100th message and we will return to the main process, which we can then use to output the desired information for each user.

Initially when trying to track friends, I went through each user, and checked if they had sent a message to each other user who had sent a message to them. However, this means that when tracking friends for the other users, I will be repeating many of the same calculations as I had not implemented any way to store previous states. To improve on this, I decided to implement memoization via a Map. The Map stores key/value pairs of the form (User A, User B) → Bool denoting if there is a friendship relation between the two users. The Map is initialized with all default values as True. To update the entries, I create an array of Bool of length users which denotes whether the current user has received a message from that other user at that index. I then fold over that array, performing a ‘logical and’ operation on the key/value pair (current User, other User) to update the status. Performing this for each user, each value can be represented as:

*True && current User messaged other User && other User messaged current User*

This correctly implements the desired functionality for modelling friends in the Map while only performing each calculation once. Finally, filtering the list of names of other users by these Map values gives the list of the current users’ friends. I looped over all the users and outputted their list of friends as comma separated values. To improve the functionality in future, I would look to try and update the friendship status of users on the fly with pairs of Bool signifying messages sent instead of checking friendship status afterwards, as this would more accurately model a social networking platform. It would also then make more sense to initialize all the map values as False, as users do not start as friends within a social network.