ECS713 Functional Programming, Fall 2023, Final Project

Chat Simulation with Concurrent Programming

Chin-Yun Yu, Centre for Digital Music, QMUL

This project utilises Haskell concurrent programming module **Control.Concurrent** to simulate ten users sending random messages to each other at random time intervals. Specifically, the following features were implemented:

- Allocate separate threads for each user to send messages
- Display how many messages each user had received and read
- Display how many messages each user had sent and been read

The above texts in blue are the extra features not given in the specifications. The program has two command line arguments where: the first is the number of users N, and the second is the maximum number of messages. Please check out the accompanying readme file to learn more about how to run the program.

Multi-Threading

My implementation uses $2N + 1$ threads, consisting of

- Main thread MT (count and forward messages) $\times 1$
- User thread UT (wait for a random interval, pick up a user and send a message) $\times N$
- Slave thread ST (receive messages and put them into corresponding data structures) $\times N$

The reason I give two threads for each user is that if only one thread is used to receive and send messages, when there is no message coming in, the thread can be blocked forever and cannot perform other tasks. It is better to separate the tasks, so I added a slave process for each user to receive messages in the background.

MVar and Chan

$N + 1$ **Chan** is created to send messages between users. The one extra **Chan** is created for message counting purposes and is shared by every user to put messages in and read by MT. The main thread monitors the messages and forwards them to corresponding users.

To share the received messages between UT and ST, **MVar** is used. Specifically, $N - 1$ **ChatBox** is created for each user to record the history of their messages with other users (excluding themselves). They are packed inside **MVar** as both UT and ST edit them, so the program uses $N(N - 1)$ **MVar** in total.

Extending the functionality: Do you copy?

Going beyond the basic functionality of just sending messages, I implement a feature that is commonly seen in modern social media platforms: **telling the message *sender* whether their messages have been read**. To do this, I made the following changes:

- Distinguish messages into two states: **read** or **unread**
- Adding an **Ack** message type from *receiver* (assuming *sender* sends regular message to *receiver*) telling *sender* how many messages has been read by *receiver*

Thus, each ST has two tasks: 1) receive a message and add it into corresponding **ChatBox**, or 2) receive an **Ack** and update the corresponding **ChatBox**. Each UT also reads every unread message from a picked random user before sending a random message to them, along with an **Ack** message saying how many new messages have been read. Fig. 1 illustrates these interactions.

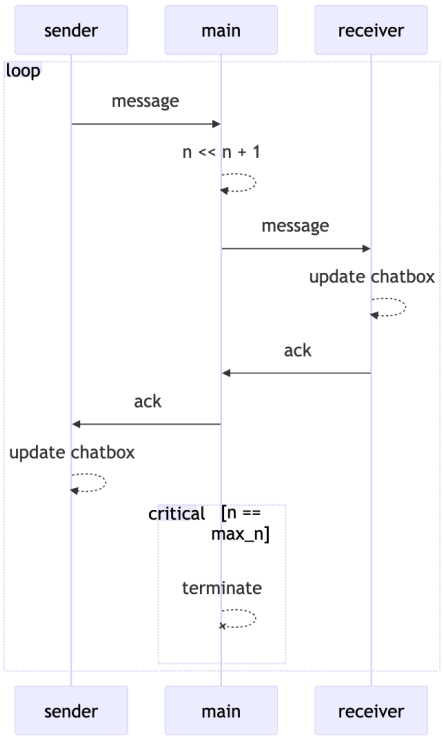

Figure 1: A simplified diagram of the message-sending process, showing the execution order of each operation. Note that both *sender* and *receiver* are run by two threads.

Data type

Each **Message** has two fields: a string of random content and a **Direction**, which is a pair of strings containing the name of *sender* and *receiver*. **Ack** contains a integer stating the number of read **Message** and also has a **Direction** field. The **User** type contains a unique string as its name and a Map object with other user names as keys and **ChatBox** as values. It is shared by one UT and one ST.

Discussions and future works

It is possible to add a new state **delivered** to the system to make it more like modern social platforms. It could be done by automatically notifying the *sender* every time ST receives a **Message**. Moreover, for simplicity, this implementation does not allow users to send messages to themselves. So far, the system has no error handling mechanism as the simulation is done purely between computer threads and has no risk of package loss. More advanced paradigms might be needed if we want to port this system to the physical world, where the hardware can cause package loss and has no guarantee always to deliver the messages successfully.