# Simulated Social Network with Software Transactional Memory

ECS713P Function Programming - Individual Coursework

Mihir Singh (220158082)
MSc., EECS
Queen Mary University of London
m.singh@se22.qmul.ac.uk

***Abstract*—**The aim of this project is to simulate a social network by emulating 10 users, each with the same behavior where they pick another user at random and send a random message to it after random intervals.

***Keywords—software transactional memory***

## I. Introduction

The main thread is handled by *spawnThreads,* and it is responsible for spawning and managing a thread for each User. It's also where a logging service is initialized to capture logs asynchronously. Each thread is responsible for simulating a User's behavior that consists of the following steps:

1. Select a random User by generating a random index for the list of all Users.
2. Check if the selected User is same as the User assigned to this thread, if so, then start again from 1.
3. Wait of a random amount of time by generating another random number.
4. Push a new message to a channel.
5. Log the event and repeat from 1.

The models for the datatypes User and Message are described in Table I.

After a total of 100 messages have been pushed to the channel, *spawnThreads* is complete and the *main* function calls *generalStats* to display total number of messages and messages received by each User.

TABLE I. Data Models

| Attribute | Type | Description |
|---|---|---|
| **User** | | |
| user_id | Int | To keep track of unique User |
| username | String | Username for display |
| **Message** | | |
| from | User | Message was sent by this User |
| content | String | Contents of the message |
| to | User | Message was sent to this User |

## II. Issues Faced

### A. Keeping a track of all Messages

To keep a track of all the messages sent, a *TChan* (of type *Message*) was used (from the *STM* Library). It acts a FIFO channel of *MVars* that helps the thread maintain mutual exclusivity when sending messages. The *TChan* follows the control mechanism of a Software Transactional Memory (STM) thus avoiding the caveats of a conventional *MVar* like race conditions and deadlocks.

A *TChan* allows us to atomically write to and read from it. The main thread (function *spawnThreads*) is blocked until it reads a 100 Messages from the channel.

### B. Logging Events (Extra Feature)

The *Logs*(.hs) module was developed to let us log events as they were happening. For the scope of this project, the event to be logged was the sending of a Message, but the *Logs* module is robust in many ways and can be used with other applications. The logging service is capable of logging events in the correct order of them occurring. It is also asynchronous, i.e., it doesn't block a thread while it interacts with I/O to log an event, instead it exists in a separate thread of its own.

When a logging service is started, a new thread is created for the logging service along with a new *TMVar*. The *TMVar* helps ensure that the events are logged in the correct order. The logging service thread is blocked while waiting for a value from *TMVar*. Each time an event is logged, the Log is written to the *TMVar*, thus unblocking the logging service, and handling the event. An extra *TMVar* is used as a "stop marker". When this "stop marker" is available (empty value written to *TMVar*), the thread is considered complete and thus stopped. The *"Event"* datatype is to handle the type of events, and *"Log"* datatype is described in Table II.

* Even though the Logs are outputted to STDIO, they can be saved in a file as well (since I/O action is atomic due to *TMVar).*

TABLE II. Log Data Model

| Attribute | Type | Description |
|---|---|---|
| sourceUser | User | Source of event to be logged |
| event | Event | Type of event to log |
| timestamp | UTCTime | Timestamp in UTC |