INDIVIDUAL PROJECT REPORT

Simulation of a social network

Report By : Nishant Kumar

INTRODUCTION: In this project I have implemented a concurrent simulation of social network using Haskell, The goal of this project is to simulate a social network where users can send random messages to other users in parallel, in simpler words here we are managing the shared resources. This project has helped us learn the fundamentals of Functional Programming through the use of Haskell Programming Language and its concepts of Concurrency, Threads , Parallelism, Monad etc. In this project I have tried to implement the concept of **SOLID principle**, wherever it suits well.

What task does it perform?

This application is built using Haskell and it is a concurrent multi-threaded simulation of social network, It simulates 10 users ,each user is running on their independent thread using **"forkIO"** , which simulates asynchronous communication in this social network application. Each user sends random messages to others and after the number of messages reaches 100 messages then the application stops and prints the output , here I have used Monads and MVar too. I have also implemented a custom SocialLife interface and followed **SOLID principles**.

Methodology and Implementation -

i) Core Simulation: The base of this project is , it uses the concepts of threads which behaves independently for
User communication.Here multiple users send messages using threads without waiting for others to complete their own process. In this each user runs on their own thread which helps us to use the concept of asynchronous. In this project I have used **"forkIO"** for thread ,in addition to this I have used MVar to safely share data between user threads. I could have used **"STM (Software Transactional Memory) in place of MVar"** for performing multiple actions as a single step as it guarantees Atomicity (either everything updates or nothing changes) which is the core of **ACID Principle** and it also provides no deadlock but it adds a little complexity,and also MVar is faster for single variable updates.

ii) Data Structure and Types:
I have created **Types.hs** to model the data clean and structured, I have created 3 data types here,
a) User: It defines username of the **username**, the **inbox** (list of messages received), **messageCount** (how many message received), **sentCount** (how many messages successfully sent) and battery (battery percentage)
b) Message: It defines **sender**(The username of the sender), **content**(The text content of the message), **topic**(The topic of the message (Extension).
c)Topic: it defines **topics** in social media like Sports,Tech ,Random, Music,News.

iii) Battery life (creative feature): I have implemented the feature of **"Battery life"** to give a touch of real life scenario ,In this every users starts with a random value of battery percentage and after every send message it will drain the battery percentage of **5 %** and once the users reach to **0 %** then he/she becomes offline and will not able to participate in chat. This will add some kind of uncertainty as in the whole chat some user might go off early and some will dominate the conversation.

iV) Parallel Data Analytics (Extra Feature): I have counted the **total word of every message** which was received by each user, but to make it more optimized and counting them one by one which makes it slow i have used the concept of **"Parallelism"** , it thus helps in distributing the task between all available CPU cores. I have used "parMap" and "rpar" which helps in distributing the counting task.

V) Statistics: I have added **3 main features** to this project implementation namely **:**

a) Stats of Users : in the output we can see the user , the number of messages sent by the particular user , the number of messages received by the particular user and Total battery left after the conversation completed , if its 0 it means the battery finished .
b) Analysis the Trending Topics : In this it shows what the user talks about and it mentions the topics with total number of them mentioned in the chat ,this adds a degree of randomness in our topic **.**
c)Total word exchanged : At last this value shows the total number of words exchanged by the individual use.

Additional Implementations:

i) Exception handling: I have used the concept of error handling in such a way so that this project becomes resilient and doesn't breaks in uncertain scenario, I have isolated each user thread so that if it fails it doesn't affect other threads and also added a **"Safety timeout"** so that the deadlock condition would be prevented if the network gets too occupied . I have also used error handling for message.log so if the message.log file is not accessible then the simulation will detect this and handle logging failure(wrapped logging in try/catch).
ii) Output Formatting: I have made sure the output formatting should be implemented properly on the terminal to get more clarity and increase readability. Instead of just showing raw numbers I have tried to implement and show proper structured tables. The tables are made using unicodes and it automatically finds the column width based on the content.

Challenges and Solutions:

i) Race condition handling: As there are 10 users with their own threads , it was causing issue at the beginning as all of them try to log into log file or trying to update the global message counter at the same time, it might have caused the text to be mixed in log or inaccurate count, so to handle this i have used **"MVars" ,** this helps to handle our previous conflict ,, now the user which hold the Mvar can write to the file or update the counter score , everyone else has to wait for thrift turn.

ii) Coordinating Threads: As all the threads started , it is difficult to stop everyone at the same time after completion of 100 message conditions mentioned in the project requirement, as it might be the case where few threads can run in background. So to solve this I have created a **"Done Signal"** , an empty Mvar. As the message reaches 100 count the main program waiting for this specific signal fills the Mvar and instantly alerts to shut down the simulation.

iii) Randomness: Initially the User code was written in strictly hardwired, lets say if we wanted to change how we will run the simulation, we will have to rewrite the user code, to solve this I have added the concept of "Interfaces,now if the user want to do any functionality such as "sending the message" or "checking the Battery".
It doesn't care about how it works, it just trusts the environment Env.hs, This separation makes the code robust and much cleaner.

How to Run the Application -
This project uses Stack for building and running the application. The following commands describe the key operations:

1: Build the Project: Use "stack build" to build the project

2:Run the Project: Use "stack run" to run the project

After that you will be provided by output on the terminal which will look like the below :

```
Application "social network" Staring ->

 note that: If the user's battery drains to 0%, User cannot send any more messages.

Simulation limit reached (100 messages).
```

```
--- Stats of Users ---
```

User	Sent	Received	Battery Left
User1	11	7	6%
User2	7	17	0%
User3	12	9	2%
User4	10	9	7%
User5	9	9	52%
User6	10	10	32%
User7	11	11	63%
User8	10	6	1%
User9	8	9	28%
User10	12	13	46%

```
--- Analysis the Trending Topics  (This is extension of the project) ---
```

Topic	Mentions
Music	26
Tech	23
News	20
Sports	17
Random	14

```
---- Total Words Exchanged ----
```

User	Total Words
User1	49
User2	119
User3	63
User4	63
User5	63
User6	70
User7	77
User8	42
User9	63
User10	91

Module Structure:

i) Interfaces.hs : Definition of SocialLife Interface.

ii) Types.hs : It defines the type of “User,”Message” and Topic.

iii) Env.hs : It implements the “SocialLife interface” for the SocialM monad. It also handles ‘IO’, ‘Mvar’ and thread management.

iv) User.hs : This module involves business logic. It defines what a user does while using the “SocialLife: interface.

v) Main.hs : This is the entry point of the application, it initializes the state,spawns threads and handles exceptions and also prints the final output.