CITS3242 Project Report – Part 2

Author: Ash Tyndall, 20915779. Partner: Nic Barbaro, 20502247.

Our solution to part 2 uses an asynchronous message passing structure to permit the communication and negotiation of laboratories between clients with no lab explosions. We constructed a system that used the guarantees of the F# MailboxProcessor library to ensure that we can perform actions concurrently without any chance of deadlock.

Unfortunately due to a lack of time, our project has some deficiencies, which will be discussed along with its general structure below.

Asynchronous Design

The design of the project used five meaningful states, intending to harness the way the project needed to react to the given messages under a variety of circumstances.

States

Idle: The client is not currently doing anything, or has just received something to do and will transition states.

Waiting: The client has an experiment, and is currently contacting other clients to get a lab to run the experiment on.

Busy: The client is currently doing an experiment and waiting the result.

Gifting: The client is attempting to transfer a lab to another client, and is delaying the response to messages regarding labs until the transfer is complete.

Terminated: The client has terminated and will not respond to future messages.

Messages

OwnLab: Asking who owns the given lab. Receiving client replies with themselves if they own the lab, or asks the client they have listed in their last known coordinates.

WantToDo: Asking to perform an experiment on the receiving client's lab. Receiving client replies with "I have queued it", "It is already queued", "I have no lab" or "I will gift you my lab".

NowOwn: Informing the client that they can now take possession of the given lab and client list. Receiving client can either reply with Accept or Decline.

ExpComplete: Posted by the client to inform itself and other clients of the completion of their experiments as sufficiencies of some experiment.

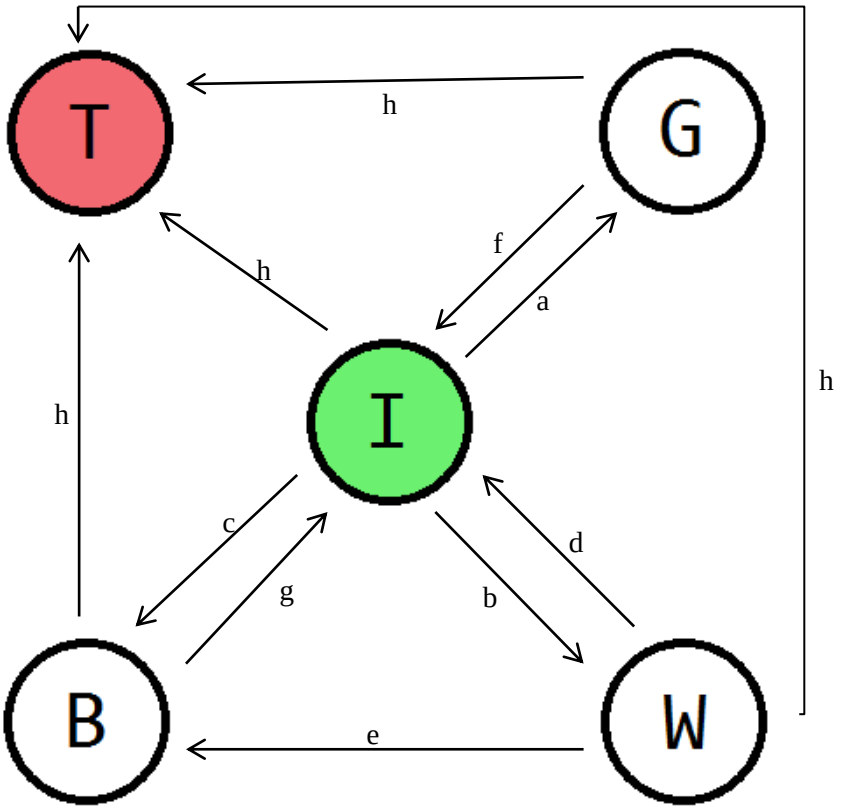
GiftComplete: Posted by the system when it receives a response to a NowOwn message. Client then reacts to the acceptance or declining of the lab transfer, processing queued messages.

ARplyWantDo: Posted by the system when a WantToDo message response is received by the client.

ARplyExpDone: Posted by the system the lab ExpDo finishes executing. Passes back the result and the experiments it sufficed for. The system then sends a message to each involved client.

State Transitions

Below is a state transition diagram, demonstrating the circumstances in which the project will change its state. The green circle represents the entry state (the state the solution begins in) and the red circle represents the terminal state (the state the solution should move to when complete).



a. Client has a lab, and there are clients queued for it.
b. Client has a lab, there are no other clients queued for it, and it has queued experiments but no lab.
c. Client has a lab, there are no other clients queued for it, and it has queued experiments and a lab.
d. Client receives word that its experiment has been completed by another lab.
e. Client negotiates a lab from another client.
f. Gifting process ends, and the client has either transferred lab ownership or has not but has no more experiment to run.
g. Client completes experiment.
h. Termination message is received.

Interesting Features

I think one of the most interesting features of our project is the Gifting state, as it provides an interesting approach to the handling of lab transfer in an asynchronous environment.

The core idea of the Gifting state is that when there is potential for a lab to change hands, we don't want to answer any queries regarding who currently owns that lab. If we do answer such queries, we have a potential to develop information inconsistencies.

Instead what we do is queue messages that we don't have an accurate answer for yet, and wait until the client we are attempting to transfer the lab to accept or declines the transfer. Depending on their answer, the queue will then be iterated through, with each message being answered appropriately taking into account the transfer result.

Program Performance and Caveats

Our solution unfortunately does not successfully handle the completely random cases generated by the random testing system. There are several subtle bugs present in the code that cause early termination that we were unable to rectify by the project deadline.

To ensure that in our theoretical system any malformed or unexpected state changes do not lead to a hypothetical lab explosion, our code throws exceptions when one of these inconsistencies arises. However it should be noted that even when the inconsistencies run unchecked, no labs explode in all tested cases.

We have noticed that due to some subtle timing or logic issues present in our asynchronous updating of the lab's last known owners, this information can become inconsistent and propagate throughout the system. This can cause some "deadlock" like behaviour in which clients stop attempting to communicate with each other as they cannot successfully locate a lab after talking to every client. When a client in possession of a lab relays an OwnLab message between two other clients that reflects inaccurate information regarding the owner of the lab, an exception is thrown.

Another error related to this timing issue is a client may enter the busy state, and subsequently have their lab ownership information changed by the asynchronous callback (prior to the above exception being implemented). This caused unpredictable behaviour with the hasLab() function, which based its true/false return value on this ownership information.

To demonstrate that our solution does work in some cases, we have included an uncommented test case that is functional.

Program Development

As with part one, due to the difficulty in division of responsibility in this project, pair-coding was used for the majority of the program development.

When we initially decided on the use of asynchronous message passing, we sat down and designed the state diagram that is included in this report. This state diagram, and a more in-depth pseudo-code description of what each state did, helped massively in the design of a solution that had no core architectural problems.

Reflection

Part two was very different to part one in terms of the style of thinking involved. Part one focused on more core functional aspects, such as recursive construction and pattern matching, and it was the broad strokes of that construction that we had difficulty with. In contrast, part two had more straightforward core architecture, and it was the minutiae that presented the most difficulty.

Our primary downfall in that sense was not allocating enough time for debugging of the asynchronous interactions, as we did not realise that the same timing subtleties that are present in a lock/monitor approach would be found in an asynchronous message passing implementation.