Jack Cannings
jc01425
6497204

Distributed Systems – COM3026 Coursework Assignment

Student: Jack Cannings

Username: JC01425

URN: 6497204

Contents

Originality Declaration

I confirm that the submitted work is my own work (*This report as well as source code*). No element has been previously submitted for assessment, or where it has, it has been correctly referenced. I have clearly identified and fully acknowledged all material that is entitled to be attributed to others (whether published or unpublished) using the referencing system set out in the programme handbook.

I agree that the University may submit my work to means of checking this, such as the plagiarism detection service Turnitin® UK and the Turnitin® Authorship Investigate service. I confirm that I understand that assessed work that has been shown to have been plagiarised will be penalised.

Signature:

Jack Cannings - jack cannings

Jack Cannings
jc01425
6497204

Paxos Pseudocode

Implements:

Paxos, **instance** *P.*

Uses:

BestEffortBroadcast, **instance** *beb;*
EventualLeaderDetector **instance** Ω**.**
PerfectPointToPointLinks **instance** *pl.*

Upon event <*P, Init* | name, participants, upper_layer> **do**

name := name;

participants := participants;

upper_layer := upper_layer;

ballot := 0;

accepted_ballot := $\perp$;

value := $\perp$;

accepted_value := $\perp$;

trusted_process := $\perp$;

chosen := FALSE;

prepare_broadcast := FALSE;

locked := FALSE;

prepared_procs := $\emptyset$;

accepted_procs := $\emptyset$;

prepared_counter := 0;

round := 1;

Upon event <*P, Propose* | v> **do**

Value := v

Upon event < Ω*, Trust* | p> **do**

trusted_process := p

Jack Cannings
jc01425
6497204

```
Upon event <beb, Deliver |[DECIDE, value]> do
        If chosen != TRUE do
                chosen := TRUE
                trigger <pl, Send | upper_layer, [DECIDE, value]


Upon event < beb, Deliver |[PREPARE, p, b]> do
        if b > ballot do
                ballot := b
                trigger <pl, Send | p [PREPARED, ballot, accepted_ballot, accepted_value]>
        else
                trigger <pl, Send | p [NACK, b]>

Upon event <pl, Deliver | [PREPARED, b, a_bal, a_val]> do
        If b == ballot && locked != TRUE do
                Prepared_procs :=  Prepared_procs ++ {a_bal, a_val}
                Prepared_counter := Prepared_counter + 1

Upon event <beb, Deliver | [ACCEPT, p, b, v]> do
        If b >= ballot do
                Ballot := b
                Accepted_ballot := b
                Accepted_value := v
                trigger<pl, Send | p, [ACCEPTED, self, Accepted_ballot]>
        else
                trigger <pl, Send | p [NACK, b]>

Upon event <pl, Deliver | [ACCEPTED, p, b]> do
        if b == ballot do
                Accepted_procs := Accepted_procs ++ {p, b}

Upon even <pl, Deliver |[NACK, b]> do
        Locked := FALSE
        Prepared_procs := Ø
        Accepted_procs := Ø
        Prepared_broadcast := FALSE
        Prepared_counter := 0


Upon Leader == Self && Prepare_broadcast == FALSE && value != nil && chosen == FALSE && locked
== true do
        Prepared_broadcast := TRUE
```

Jack Cannings
jc01425
6497204

Prepared_counter := *Prepared_counter* + 1
Ballot := *generateNewBallot()*
trigger <*beb,* BROADCAST | [PREPARE, *Self, ballot*]>
Prepared_procs := *Prepared_procs* + [*ballot*, *value*]

Upon *Leader == Self* ***&&*** *Prepared_counter > (***S***/2)* ***&&*** *locked == FALSE* ***&&*** *chosen == FALSE* ***do***
*(Where **S** is the number of participants)*

If max a_bal in *prepared_procs* != *ballot* **do**
Value := a_val with max a_bal in *prepared_procs*
Locked := TRUE
Else
Locked := TRUE
trigger <*beb,* BROADCAST | [ACCEPT, *Self, ballot*, *value*]>
Accepted_procs := *Accepted_procs* ++ [*name, ballot*]

Upon *Leader == Self* ***&&*** **|***Accepted_procs***|** > (**S**/2) && *chosen* == FALSE **do** *(Where **S** is the number of participants)*

trigger <*P* | [DECIDE, *value*]>
trigger <*beb,* BROADCAST | [DECIDE, *value*]>

Upon *generateNewBallot( )* **do**

Round := *round + 1*
newBallot *= round * **S** + **I*** *(Where S is the number of participants, and **I** is index of name in participants)*
if newBallot > *ballot* **do**
ballot := newBallot
else
generateNewBallot( )

Jack Cannings
jc01425
6497204

Uniform Consensus Property Satisfaction

Termination

Due to the properties of Eventual Leader Detector, there will eventually be a time in which no two correct processes trust a different correct process. When this happens, the value which the leader is attempting to decide upon will be accepted by the all correct processes. Once this has happened, the leader will decide on its value and *beb broadcast* that *decide* message and all correct processes will deliver it, which is ensured by the correctness of PerfectPointToPointLinks. Upon delivering the *decide* message from the leader, each process will *decide*, terminating the program.

Validity

Validity follows from the algorithm that each process is initiated without a value, and only values that have been proposed from the upper_layer can ever be chosen. Once the leader decides on its value, it broadcasts that value to all other processes, which will then decide on that same value.

Integrity

We can see from the pseudocode on the right, that upon deciding a value, each process changes the *chosen* variable to true, thus preventing a second decision to be made by a correct process under any circumstance.

Upon event <*beb,* DECIDE | value> **do**
    If *chosen* != TRUE **do**
        chosen := TRUE
        trigger <*pl*, *Send* | *upper_layer*, [DECIDE, value]

Uniform Consensus

From the algorithm, it is clear than the only scenario in which a leader will *decide* on a value is if it has delivered an *accepted* message from a majority of correct follower processes, and upon deciding on the *value **v*** it will broadcast that decision to all correct processes through *beb*. During the process of having the value **v** accepted by the follower processes, each follower process would have adopted the leader's *value **v*** into *accepted_value,* along with the *ballot **b*** into *accepted_ballot*. In a scenario where the leader decides on a value **v**, but fails before broadcasting the *decide* message to the rest of the processes, any new leader that is elected by Ω will be able to discover the value **v** accepted by the follower processes which was decided upon by the previous leader. For this reason, any *value* **v'**

Jack Cannings
jc01425
6497204
which could be decided by the new leader, **v' = v** holds true, due to the new leader adopting the value **v'** associated with follower processes' largest *accepted_ballot* **b**.

Pseudocode Translation

In this section, I will list some of the most important parts of the pseudocode and show examples of how they have been translated into elixir code.

Upon event <*P, Init* | name, participants, upper_layer> **do**

The initialisation event detailed in the pseudocode is executed using two function in elixir. The first of the functions is the ***start*** function, which takes the arguments ***name, participants*** and ***upper_layer***. This function registers each of the processes, so they are ready to be initialised with a state. Next is the ***init*** function, which takes the same arguments as above, and sets the default state of each of the processes.

```
defmodule Paxos do
    def start(name, participants,upper_layer) do
        pid = spawn(Paxos, :init, [name, participants,upper_layer])
        case :global.re_register_name(name, pid) do
            :yes -> pid
            :no  -> :error
        end
        IO.puts "registered #{name}"
        pid

    end
    def init(name, participants,upper_layer) do
        state = %{
            name: name,
            participants: participants,
            upper_layer: upper_layer,
            ballot: 0,
            accepted_ballot: 0,
            value: nil,
            accepted_value: nil,
            trusted_process: nil,
            chosen: false,
            prepare_broadcast: false,
            locked: false,
            prepared_procs: %{},
            accepted_procs: %{},
            prepared_counter: 0,
            round: 1,


        }
        run(state)
```

```
def run(state) do
    state = receive do
        {:propose, value} ->
            state = %{state | value: value}
            state

        {:trust, p} ->
            state = %{state | trusted_process: p}
            check_internal_events(state)

        {:decide, v} ->
            state = if state.chosen != true do
                state = %{state | chosen: true}
                send(state.upper_layer, {:decide, v})
                state
            else
                state
            end
            state
```

Upon event <*P, Propose* | v>

Upon event < Ω*, Trust* | p>

Upon event <*beb, Deliver* |[DECIDE, value]>

Upon event < *beb, Deliver* |[PREPARE, p, b]>

Upon event <*pl*, *Deliver* | [PREPARED, b, a_bal, a_val]>

Upon event <*beb, Deliver* | [ACCEPT, p, b, v]>

Upon event <*pl*, *Deliver* | [ACCEPTED, p, b]>

Upon even <*pl*, *Deliver* |[NACK, b]>

The function ***run(state)*** in the elixir code is the function which handles all the events above. Each of these events is treated as a message which contains the event name, and arguments, this is in the format of ***{:event, arg_1, … arg_N}.*** Each time a process receives a message, it will be processed by the ***receive do*** statement in the ***run(state)*** function.

Jack Cannings
jc01425
6497204

All the *accepted_ballot*, *accepted_value* etc… data from :*prepared,* and *:accepted* is handles using **Maps** in elixir. The **Map** data structure is a *key-value* type data structure in elixir, this makes it easy for me to be able to search for the max *accepted_ballot* stored, using the functions from the **Map** library as well as from the **Enum** library.

```elixir
# Send message m point-to-point to process p
    defp unicast(m, p) do
        case :global.whereis_name(p) do
                pid when is_pid(pid) and pid != self() -> send(pid, m)
                pid when pid == self() -> :ok
                :undefined -> :ok
        end
    end

    # Best-effort broadcast of m to the set of destinations dest
    defp beb_broadcast(m, dest), do: for p <- dest, do: unicast(m, p)
```

The ***broadcast*** events throughout the pseudocode are all handled by an implementation of *best-effort-broadcast,* which in the source code, is handled by the *beb_broadcast()* and *unicast()* functions. The *beb_broadcast()* function takes arguments ***m*** for message, and ***dest*** meaning the set of processes to send to. It then triggers the *unicast()* function, which takes the message ***m*** which contains the format of message detailed above, and ***p,*** which is each of the processes in ***dest. ***Unicast()* also takes care to not send a message to the original source process of the broadcast call.

Upon *Leader == Self* **&&** *Prepare_broadcast…*

Upon *Leader == Self* **&&** *Prepared_counte*r…

Upon *Leader == Self* **&&** **|***Accepted_procs***|**…

The three events above (paraphrased) from the pseudocode are all handled in the ***check_internal_event(state)*** function. These events can only be executed by trusted functions, and they handle the quorum checks, various broadcasts and state changes. The ***check_internal_events*** function is called in many of the message event handlers, to make sure that the leader process can react to new messages being received.

```elixir
def check_internal_events(state) do
    #Leader to broadcast
    state = if state.name == state.trusted_process and state.prepare_broadcast == false and state.value != nil and state.chosen == false and state.locked == false do
        state = %{state | prepare_broadcast: true, prepared_counter: 1}
        state = generateNewBallot(state)
        beb_broadcast({:prepare, self(), state.ballot}, state.participants)
        %{state | prepared_procs: Map.put(state.prepared_procs, state.ballot, state.value)}
    else
        state
    end

    #Upon a majority quorum of prepared received
    state = if state.name == state.trusted_process and state.prepared_counter > trunc(length(state.participants)/2)and state.locked == false and state.chosen == false do
        state = if Enum.max(Map.keys(state.prepared_procs)) != state.ballot do
            %{state | locked: true, value: Map.get(state.prepared_procs, Enum.max(Map.keys(state.prepared_procs)))}
        else
            %{state | locked: true}
        end
        beb_broadcast({:accept, self(), state.ballot, state.value}, state.participants)
        %{state | accepted_procs: Map.put(state.accepted_procs, state.name, state.ballot)}
    else
        state
    end

    state = if state.name == state.trusted_process and length(Map.keys(state.accepted_procs)) > trunc(length(state.participants)/2) and state.chosen == false do
        send(self(), {:decide, state.value})
        beb_broadcast({:decide, state.value}, state.participants)
        state
    else
        state
    end
end
```

These event handles are controlled by **if else** statements, which only allow the leader processes to continue if they have met the conditions expressed in the pseudocode.

Jack Cannings
jc01425
6497204

To ensure ever increasing unique ballots which don’t conflict with other leader processes, the ***generateNewBallot(state)*** function is used. This function alters the state, and calculates the new ballot using the following formula: **i * S + P**

*Where **i** is the state.round variable, **S** is the length of the state.participants variable, and **P** is the index position of the process calling the function in the participants List.*

```
def generateNewBallot(state) do
    state = %{state | round: state.round + 1}
    newBallot = state.round * length(state.participants) + Enum.find_index(state.participants, fn(x) -> (x == state.name) end)
    state = if newBallot > state.ballot do
        %{state | ballot: newBallot}
    else
        generateNewBallot(state)
    end
    state
end
```

There is then a check to make sure that the new ballot is indeed larger than any other that the process may have adopted from receiving *:prepares* from other leader processes, if this is not the case, it will run the function once more, with an increased *state.round* integer.