1a)

A perfect failure detector is a failure detector that works by sending a ‘heartbeat’ message to every other process using a perfect peer-to-peer link. If it doesn’t receive a reply before a timeout, the process that didn’t reply is considered to be crashed and is suspected forever on from then.

Safety:

Strong accuracy - no process is suspected before it crashes.

Liveness:

Strong completeness - every process that crashes will eventually be permanently suspected by every correct process .

1b) (I haven’t tried to run this)

defmodule select\_correct\_process

def start(component, [p|\_]=processes) do

send component, {:ls\_leader, p}

next(component, p, correct\_processes, suspect\_processes)

end # start

defp next(component, leader, correct\_processes, suspect\_processes) do

receive do

{:suspect, suspect} ->

[p|rest] = List.delete(correct\_processes, p)

if leader == suspect do

send component, {:ls\_leader, p}

end

next(component, [p|rest], Enum.add(suspect\_processes, p))

end #receive

end

end #module

defmodule perfect\_failure\_detector

def start(selector, pl, processes) do

send\_heartbeat(selector, pl, processes)

end

defp send\_heartbeat(selector, pl, processes) do

for p <- processes do: send pl, {:pl\_send, p, {:heartbeat, pl}}

next(selector, processes, [], Time.time+TIMEOUT)

end # send\_heartbeat

defp next(selector, pl, processes, received, timeout\_time) do

if Time.time <= timeout\_time do

suspect = Enum.filter(processes, fn(p) -> !Enum.member(received, p))

send selector, {:suspect, suspect}

send\_heartbeat(selector, pl, received)

end # if

receive do

{:pl\_deliver, {:response, p}} ->

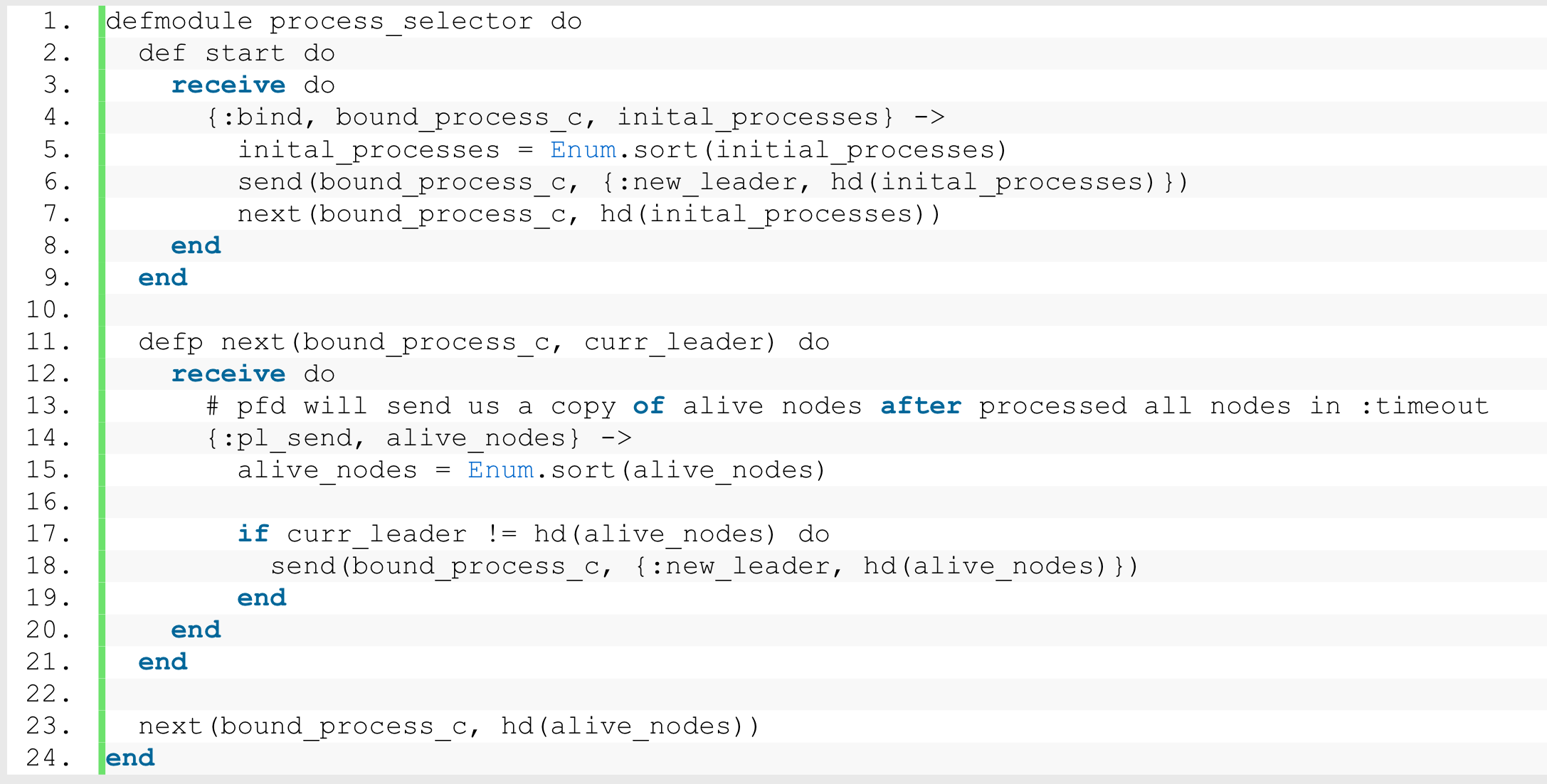
next(selector, pl, processes, received++p)

end # receive

end # next

end #module

Another way



c)

It satisfies the safety property strong accuracy because the use of the PL component (perfect peer-to-peer link) ensures that all correct processes receive and reply to the heartbeat message.

If satisfies the liveness property strong completeness because any process that doesn’t reply to the heartbeat within the time limit is suspected forever.

The performance of select\_correct\_component is in best case O(1) since it never has to change the leader if no process crashes. Worst case O(N) where N is number of processes since it has to change leader N times.

However, the perfect failure detector has to send the heartbeat message to every correct process on every interval of TIMEOUT. …

It satisfies the accuracy property of the select\_correct\_process by the usage of PFD.

And it satisfies the eventual selection property since we will always return a result when there is at least one correct processes.

2

defmodule pb\_broadcast do

start(R, K) do # R rounds, K random processes

receive do

{:bind, C, LL, Processes} -> next( R, K, C, LL, Processes, MapSet.new())

end

end

next( R, K, C, LL, Processes, delivered) do

receive do

{pb\_broadcast, M} ->

MapSet.put(delivered, M)

forward(LL, {pb\_data, M, R, self()}, K)

send C, {pb\_delivered, M}

{ll\_deliver, From, {pb\_data, M, S, sender}} ->

delivered =

if MapSet.contains(delivered, M) then

delivered

else

send C, {pb\_delivered, M}

MapSet.put(delivered, M)

end

case S > 0 of

true -> forward(LL,{pb\_data, M, S - 1, sender}, K)

false -> skip

end # case

next(R, K, C, LL, Processes, delivered)

end # receive

defp forward(LL, msg, K) do

for p <- random\_processes(Processes, K) do:

send LL, {:ll, p, msg}

end

defp random\_processes(Processes, K) do

# returns K randomly selected processes excluding self

end

end

b)

Performance – in the first round, K messages are sent, and afterwards each of the K processes sends K more messages, so K^2 are sent in the second round, K^3 in the third. Therefore total K + K^2 + ... + K^R = O(K^R) messages are sent, not accounting for message loss. Hence, increasing K and R increases message complexity. At a practical level, if K and R are sufficiently large such that each process is chosen repeatedly in several rounds, this can decrease the throughput of broadcast messages from the client by increasing the number of ll\_deliver messages in the message queue. Increasing R increases time complexity as each message broadcast then incurs R steps of the algorithm. “Locally”, increasing K means that the algorithm spends longer inside the for-loop in the forward function. On the other hand increasing K and R improves reliability: increasing K improves the probability that a process is chosen in a given round, while increasing R mitigates the eventuality that the lossy links component fails to deliver the message to chosen processes by increasing the likelihood of a process being chosen in multiple rounds.

3a)

In leader election, processes have to wait for the leader to detect a crash

to agree that a process has crashed while in group membership any process can

propose it.

In leader election, a process only knows that the leader is working while it

doesn't know the state of other processes while in group membership, it knows

the status of all processes.

b)

Completeness: If a process p crashes, then eventually every correct process will install

a view which does not have p as its member

Follows from perfect failure detector, so if a process has crashed, eventually

every process will detect that it has crashed so eventually every proposal won't

include the crashed process.

Then from the validity of the consensus, if a view is decided on, it must have

been proposed so the view won't include the crashed process.

Accuracy: If a process installs a new view which does not contain a process p, then

process p must have crashed

Follows from perfect failure detector as a process only proposes a process if it

has detected it has crashed via the pfd.

c)

Assuming rounds(i.e. we know the diameter of the network) exist and each process has a UID.

Each round, each node broadcasts the highest UID it has found to

all of its neighbours. For the first round broadcast your own UID.

Number of rounds is the diameter of the network and then after that many rounds,

if the max UID is your own, declare yourself as leader.

Whenever a node receives a new UID from a broadcast, it compares it with its own

UID\_max and updates UID\_max if the new UID is larger.

d)

No.

If 2 processes, A, B, claim that they're leader and broadcast that they are

leader. Then A will receive B's claim that B is leader and as there's no UID, A

will think that this is its own claim that it's leader. If A can distinguish

the fact that it's not its own claim and that it is B's claim, then the

processes aren't anonymous. Then B will do the same for A's claim so B will

think that A's claim is its own claim. As it's a ring, every other node would

have received at least one claim, A or B's and assume that a leader has been

nominated. So both A & B will think their leaders.

Now if you replace, the claim that they're leader with any other message, that

is used in any leader election algorithm, it is clear how a leader can't be

nominated.

As 2 processes may broadcast the same step of any leader election algorithm at

the same time. Then it would receive the other processes message at that step,

assume it is its own message and continue on with the algorithm.

So 2 processes can end up leader at the same time.

Randomness can't be used to decide the leader else it's not deterministic.

e)

In the best case, the ring is in ascending order so each node will message its

neighbour which has a UID greater than it and then neighbour won't forward it

on. This is for all nodes apart from the node with the largest UID which is

before the UID with the lowest value where the UID will be forwarded around by

every node.

So, theres n messages in the first round and then 1 message per round for the

remaining n-1 rounds as UID\_max is sent around the network. So it's O(n).

For worst case, the ring is in descending order. Then a for node i, its UID will

be sent i times. As every node in front of is less than it, it will get forwarded

around the ring until it reaches the start of the ring where it gets stopped by

the node with UID\_max. So node 1 sends 1 message, node 2 causes 2 messages, node

3 causes 3 and so on. So it's 1 + 2 + ... + n and hence O(n^2).

For average case, in any ring order,

every node sends at least 1 message but then the chance that a node causes 2

messages is 1/2 when averaged over all nodes as the chance that a give node is

larger than another node is 1/2. Then for it causing 3 messages, it's 1/3. Think

of it as randomly selecting 3 nodes, what is the chance that a certain node is

the largest of the three, it's 1/3. This is then generalised to 1/k causing k

messages.

So for a single ring, the expected number of messages, a node causes is 1 + 1/2

+ 1/3 + ... + 1/n. Then multiple this by n as there's n nodes in the ring to get

n(1 + 1/2 + ... + 1/n).

1 + 1/2 + ... + 1/n is the harmonic function which grows approximately log(n) so

the expected number of messages is O(n\*log(n)).

And for declaring that a certain node is leader, it takes n messages but that's

O(n\*log(n) + n) which is still O(n\*log(n)) so we can ignore it.

For anyone looking at the tutorial solutions for this answer, for the average

case, they decide to multiply every step by n! as there's n! possible rings and

then they divide by n! to get the average of the rings.

4.a)

Iterative look-ups can be optimised to do multiple look-ups at once so when

querying a node if it has a query, you can query that node for all of your

queries.

Recursive look-ups have to do multiple searches as it doesn't know what it's

searching for for the second query before it has the result for the first

look-up.

b)

The overlay isn't correlated with the physical connections of the network so if

a node is neighboured to a node in the overlay, it doesn't mean that they are

physically close to each other so it can take a long time/a large number of

hops to communicate between neighbours of the overlay.

c)

Easily scalable

Programme portability due to common programming interface

Simplifies tasks for programmers as there's no need to worry about communication

primitives

d)

Progess:

As each process has UID and the UID's are a total order, it means that either a

process with the lowest timestamp will enter and progress is made or the process

with the smallest UID will enter and hence progress.

Bounded waiting:

Assume a process has obtained a ticket then look at what the other processes

could be doing. If they aren't already in the queue, then when they join, they

will choose a timestamp greater than the original process so the original

process will get served before them.

If they are in the queue already, then they will enter the

critical section and then join the group of processes that join after the

original process. So when they rejoin their timestamp will be greater than the

original timestamp so the original process will get served before them.

The final option is that a a process was obtaining a ticket while the original

process was so they have the same timestamp. In this case, if the process has a

lower UID than the original process, it will become as if it was already in the

queue when the original process obtained it's ticket as it will be served first.

If the process has a higher UID, then it's equivalent to a process that joined

the queue after the original process obtained its ticket. Either way, the wait

is bounded.

e)

In the first flood, N nodes would be reached. Then each of those nodes would

reach N of its own neighbours but one of its neighbours would be the original

node so each node would reach N-1 new nodes. Now if we assume that each node

that is a neighbour of the original node isn't a neighbour with any other node

that is a neighbour of the original node, each node will reach N-1 new distinct

nodes so N \* (N-1) nodes in total for an upperbound.

Then for a lower bound, each of the original neighbours are neighbours of each

other and that's it (so a completely connected graph) then N nodes would be

reached.

N \* (N-1) + 1 if you count the original node?

Average case:

https://www.physicsforums.com/threads/number-of-nodes-as-neighbors-probability-question.711555/

f)

Every read operation should return the most recent write operation according to

global ordering.

Every process sees the same order of events.

To implement it, use total order broadcasting so for reads, the local copy

should be kept up to date so request to do a read via total order and once

granted, read your local copy and announce to other processes what you read.

For writes, again request and wait to write using total order broadcasting. When

you write, announce the new value and acknowledge the new value to other

processes.

Every time a write is delivered, update your local copy.

Every time a read is delivered, do nothing.