1.

a.

defmodule WC do

def start(n, f) do

receive do

{ :bind, c, peers } -> send\_proposal(n, f, c, peers)

end

end

defp send\_proposal(n, f, c, peers) do

receive do

{ :wc\_proposal, proposal } ->

for peer <- peers do

send(peer, { :wc\_peer\_proposal, proposal })

end

recv(n, f, c, peers, proposal, 1)

end

end

defp recv(n, f, c, peers, highest, received) do

receive do

{ :wc\_peer\_proposal, proposal } ->

highest = max(proposal, highest)

received = received + 1

if received >= n - f do

send(c, { :wc\_deliver, highest })

send\_proposal(n, f, c, peers)

else

recv(n, f, c, peers, highest, received)

end

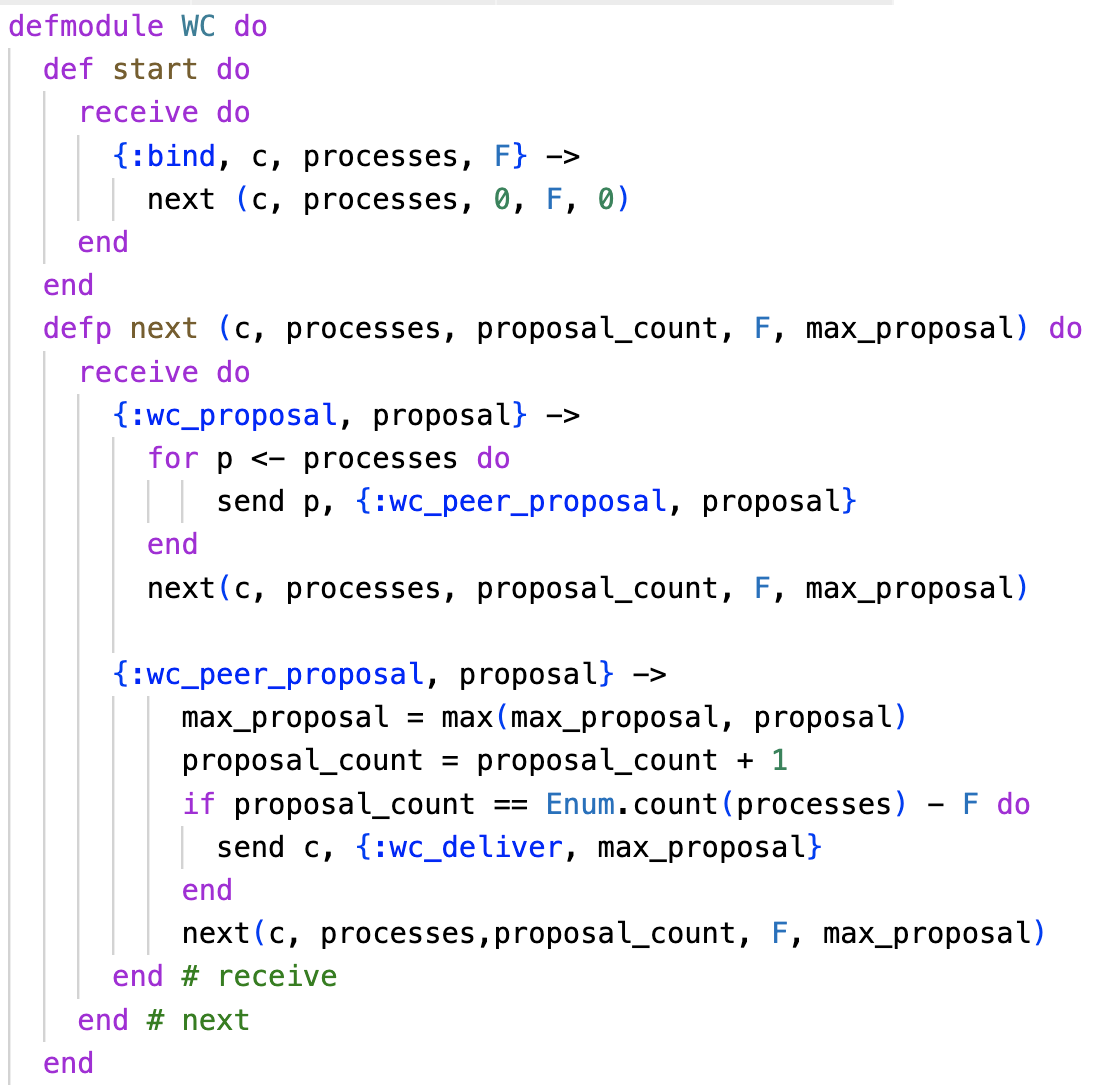
end

end

end

------------------------------------------------------------------------------------------------------------------

This is my implementation, let me know what you think!



defmodule WC do  
 def start do  
 receive do {:bind, c, processes, F} -> next (processes, 0, F, c, 0)  
 end  
 end  
 defp next (processes, proposal\_count, F, c, max\_proposal\_value) do  
 receive do  
  
 {:wc\_proposal, proposal} ->  
 for p <- processes do  
 send p, {:wc\_peer\_proposal, proposal}  
 end

{:wc\_peer\_proposal, proposal} ->

Max\_proposal\_value = max(max\_proposal\_value, proposal)  
  
 if proposal\_count == Enum.count(processes) - F do  
 send c, {:wc\_deliver,max\_proposal\_value }  
 end  
 next(processes,proposal\_count+1,faulty\_processes\_count,c,max\_proposal\_value)  
 end # receive  
 end # next  
end

b.

First two proofs are straightforward.

Last by contradiction:

V is some arbitrary value that bounds the number of distinct values which can be decided. I believe V is F + 1, so let's assume we have reached a configuration where there are at least F + 2 distinct values.

Let's take the process L which decided on the lowest distinct value. For L to have reached a decision, it would need to have processed N – F other proposals. Since L decided on the lowest distinct value, we know at least F + 1 other processes have proposed distinct higher values. If L processed a message from any of these processes, L would have decided on a value greater than what it decided on. Therefore, L did not process a message from any of these processes. This means L must have processed, at most, N - (F + 1) messages. This is a contradiction, as we know L has processed N - F proposals. Therefore, V cannot be greater than F + 1 and we've found the upper bound of V.

c. Not sure about this. I don’t believe we talked about Raft in detail.

2.

a.

defmodule VBA do

def start(c, epl, processes, n, r) do

next(c, epl, processes, n, r)

end

defp next(c, epl, processes, n, r) do

receive do

{ :vba\_broadcast, msg } ->e

for peer <- neighbours(n, processes) do

send(epl, { :epl\_send, peer, { :broadcast, 1, msg }})

end

{ :epl\_deliver, { :broadcast, cur\_round, msg }} ->

send(c, { :vba\_deliver, msg })

unless cur\_round > r do

for peer <- neighbours(n, processes) do

send(epl, { :epl\_send, peer, { :broadcast, cur\_round + 1, msg }})

end

end

end

end

defp neighbours(n, processes) do

end

End

---

Alternative:

defmodule VBA do

def start(c, epl, processes, n, r) do

next(c, epl, processes, n, r, MapSet.new())

end

defp next(c, epl, processes, n, r, seen\_messages) do

receive do

{:vba\_broadcast, msg} ->

for p <- neighbours(n, processes) do

send epl, {:epl\_send, p, {msg, 0}}

end

{:epl\_deliver, {msg, round}} ->

if !msg in seen\_messages do

send c, {:vba\_deliver, msg}

seen\_messages’ = MapSet.put(seen\_messages, msg)

end

if round < r do

for p <- neighbours(n, processes) do

send epl, {:epl\_send, p, {msg, round + 1}}

end

end

end

end

defp neighbours(n, processes) do

# returns n neighbouring processes excluding self()

# implementation not given (as stated in question)

end

end

b.

As we move into successive rounds of VBA broadcast, an increasing number of processes start rebroadcasting the message to their neighbours. In the first round, 1 process sends a broadcast to n neighbours. In the second, approximately n neighbours (not accounting for message loss) send a broadcast to n neighbours, leading to n \* n messages. In the third round, n \* n neighbours rebroadcast to n neighbours, leading to n^3 messages. This leads to a message complexity of O(n^r). For time complexity, the main factors are the time it takes for a process to send a message to n neighbours and the latency between peers. This leads to a time complexity of O(r(n \* N + R)) where N is the time it takes for a round to complete and R is the connection latency between peers.

c. Not sure about this. Is it saying that beyond the first round, further VBA broadcasts will fail? This doesn’t make sense though given what c.ii asks.

---

Possible:

Resend always unsuccessful: constant at N/P for all rounds

Resend may be successful: increasing, but plateauing, starting at N/P, horizontal asymptote at 1

3.

a.

In group membership, a consensus step is used to decide a global ordering for when each process crashes. A new consensus step is unnecessary where the group membership has not changed in composition so a consensus step would only ever be needed to agree on the next crashed process, and a single step would be used so that all participating processes agree on the order in which each process has crashed.

b.

A safety property makes a guarantee that something will never happen, while a liveness property is a property which is eventually true and allows the algorithm to make progress.

See lecture one slide 14, there is a more formal definition

c.

In a general asynchronous network, you could create a modified FloodMax which, for each round, send messages to all neighbours with the largest UID you've seen in the previous round. Once you've received a message from all your neighbours, move on to the next round until you've done enough rounds for messages to have traversed the entire graph.

Answer from tutorial:

Rounds do not exist in an asynchronous network and we would need to be able to simulate

them asynchronously.

Each process that sends a message at round R must tag the message with the round number. The

recipient waits until it has received round R message from all its neighbours before proceeding. Algorithm ends after diameter rounds.

d.

Consensus algorithms usually have some global heuristic to decide which message to accept. The global heuristic could be modified to prioritise messages with the most failures detected and modify each consensus step so that multiple processes could have failed within a single step. Then, if a process detects an additional failed process during a consensus step, that process can then reject proposed values and offer its own proposal with the additional failure to the other peers. Since all peers have their own failure detectors, they will likely agree with the proposed view and rapidly reach consensus.

e

If VariableSpeed nodes start sending messages at arbitrary times, there's the possibility of ending up in an inconsistent state where a process with a higher UID completes the loop while a late-starting lower UID begins the election process. This could be resolved by adding a second phase to VariableSpeed. Phase 1 would work the same as regular VariableSpeed. Once a phase 1 message makes its way back to the start, the starting node sends a phase 2 message through the ring to confirm itself as leader. As nodes receive a phase 2 message, they forward it through the ring without waiting, but only if it has not received a phase 1 message with a lower UID. If a node receives a phase 2 message it sent without receiving a lower phase 1 message, it becomes the leader.

f

Since nodes are highly aware of the node layout, we could use a modified version of TimeSlice. Each node could have an individual timeout based on when it would have expected preceding nodes to start the leadership selection process before it nominates itself, with a preference for lower uid nodes to be elected. Once a node starts the election process, it sends a nomination message downwards in its column. The next nodes continue forwarding until the message loops back to the node which initiated the process. Now, it sends a message rightwards and the next node starts the same column nomination process. This continues until the last column finishes nomination and the message loops back to the originating process, which can now declare itself the leader. This requires only O(n) messages.

Note: there is answer in the tutorial 2

Answer in tutorial 2 is probably not correct, the diameter of the graph is 2, use floodmax is better

4.a)

Common global time available

b)

Simplifies tasks for programmers

Programme portability due to common programming interface

Easily scalable

c)i)

Yes

P4: R(y, 0), R(x, 0)

P1: W(x, 1)

P3: R(x, 1), R(y, 0)

P1: R(x, 1), R(y, 0)

P2: W(y, 1), R(y, 1), R(x, 1)

ii)

No as P1 reads y,0 after W(x, 1) so y, 0 has to be sequentially after x,1 but

P3 reads y,1 before x,0.

iii)

Yes

P2: W(x, 2)

P3: W(x, 2)

P1: W(x, 1)

P3: R(x, 1)

d)

They have certain number of connections to ensure they can reach more nodes.

They should have a certain upper bound on processing time so messages don't get

stuck there

d)

They have certain number of connections to ensure they can reach more nodes.

They should have a certain upper bound on processing time so messages don't get

stuck there.

High availability, I.e., online for a long time

Large storage, store an index of where to find files if itself does not have it

e)

Probability that A chooses B is k/n then prob of B chooses A is k/n so prob that neither of them choose each other is (k/n)^2

As it’s an overlay network, any node can have any neighbour so B & A being related by C has no effect. On piazza they said that just cause C chooses A, it doesn’t mean that A chooses C so we have to consider both directions.

f)

Some search to get a graph of the network.

When doing so, create a maps of resource to node and node to resource.

Then when something is requested look it in the map and go to that node.

To get route to node use something like Djistkra's.

(Tutorial 2, Q19): The lowest message complexity (zero messages) would be if the resources were replicated at each node. To replicate resources, we can use convergecast followed by a broadcast or just flood the network since we don’t care about pre-processing message complexity.

(The best distributed algorithm is when you don’t have a distributed system :(