1.a)

* upper bound on process delays i.e. time taken to do a step
* upper bound on time for a message to be delivered

b) Yes by def? (Maybe state async can’t as async is unbounded?)

I think not, if we do not know the upper bound on delays. If we do know the upper bound, then yes, we just need to ping (as a PFD) with a timeout which is some multiple of the upper bound.

c)

defmoduel FPL do

def start do

receive do

{:bind, c, pl} -> next c, pl, Map.new, Map.new, []

end

end

defp next c, pl, sent\_seq\_no, next\_seq\_no, pending do

receive do

{:fpl\_send, receiver, msg} ->

sent\_seq\_no = Map.put(sent\_seq\_no, receiver, Map.get(sent\_seq\_no, receiver, 0) + 1)

send pl, {receiver, msg, Map.get(sent\_seq\_no, receiver)}

next(c, pl, sent\_seq\_no, next\_seq\_no, pendig)

{:pl\_receiver, {p, {msg, seq\_no}}} ->

pending = pending + [{p, {msg, seq\_no}}]

for {p’, msg’, seq\_no’} <- pending, seq\_no’ == Map.get(next\_seq\_no, p’, 1) do

next\_seq\_no = Map.put(next\_seq\_no, p’, Map.get(next\_seq\_no, p’, 1) + 1)

pending = List.remove(pending, {p’, msg’, seq\_no’})

send c, {msg}

end

end

End

Just see the lecture slide on FIFO broadcast, it the same.

**Alternate with the signature he gave us?**

defmodule FPL do

def start do

receive do

{:bind, c, pl} -> next c, pl, Map.new, []

end

end

# pseqno = Map pid -> seqno

defp next c, pl, pseqno, pending do

receive do

{:fpl\_send, m, dest} ->

seq\_num = Map.get(pseqno, dest, 0)

send pl, {:pl\_send, {:fpl\_data, m, seq\_num, our\_id()}}

next c, pl, Map.put(pseqno, dest, seq\_num+1), pending

{:pl\_deliver, {:fpl\_data, m, seq\_num, sender}} ->

pending = pending ++ [{m, seq\_num, sender}]

{pseqno, pending} = check\_pending(c, pl, pseqno, pending, sender)

next c, pl, pseqno, pending

end

end

defp check\_pending c, pl, pseqno, pending, sender do

case Enum.find(pending, fn {\_, seq\_no, ^sender} -> seq\_no == Map.get(pseqno, sender, 0) do

{m, seq\_no, sender} ->

send c, {:fpl\_deliver, m, sender}

Map.put(pseqno, sender, seq\_no+1)

pending = List.delete(pending, {m, seq\_no, sender)

check\_pending c, pl, pseqno, pending, sender

otherwise ->

{pseqno, pending}

end

end

d)

It uses a sequence number to ensure that the message is always delivered in order. Same performance as PL?

Good network condition: no difference

Bad network condition: PL delivers message as soon as it receives one, FPL need to wait for the missing message and then it will deliver a bunch of messages at once.

2a

def moduleFlooding\_uniform\_consensus do

def start do

receive do

{ :bind, c, beb, n, processes } ->

next c, beb, n, processes, 1, nil, MapSet.new, MapSet.new

end

end

defp next c, beb, n, correct, round, value\_decided, proposal\_set, proposer\_set do

receive do

{ :pfd\_crash, crashedP } ->

correct = MapSet.delete(correct, crashedP)

next , , correct, , , ,

{ :uc\_propose, value } ->

# you could just use MapSet.put for lines below??

proposal\_set = MapSet.union(proposal\_set, MapSet.new |> MapSet.put(value))

proposer\_set = MapSet.union(proposer\_set, MapSet.new |> MapSet.put(self()))

# proposer\_set = MapSet.put(proposer\_set, self())

# the last two steps could be simplified since :uc\_propose message starts a new

# consensus, so both sets should be empty

send(beb, {:beb\_broadcast, self(), {:uc\_data, round, proposal\_set}}

next , , , , , proposal\_set, proposer\_set

{:beb\_deliver, proposer, {:uc\_data, r, prop\_set}} when round == r ->

proposal\_set = MapSet.union(proposal\_set, prop\_set)

proposer\_set = MapSet.union(proposer\_set, MapSet.new |> MapSet.put(proposer))

Process.send\_after(self(), {:check\_if\_consensus}, 0)

next , , , , , proposal\_set, proposer\_set

{ :check\_if\_consensus } MapSet.susbset?(correct, proposer\_set) and value\_decided != nil ->

if round == n do

value\_decided = Enum.min(proposal\_set)

send(c, {:uc\_deliver, value\_decided})

next , , , , , value\_decided, ,

else

next, , , , round + 1, , , MapSet.new

end

end

end

b) Uniform agreement: *No two processes decide on different values*. This holds because the processes that reach the last round have the *proposal\_set*s equal.

Termination: *Each correct process eventually decides.* This holds because all correct processes reach the last round, when they decide on the minimal value.

c) O(N) time (because we have n rounds), O(N^3) messages exchanged between all correct processes.

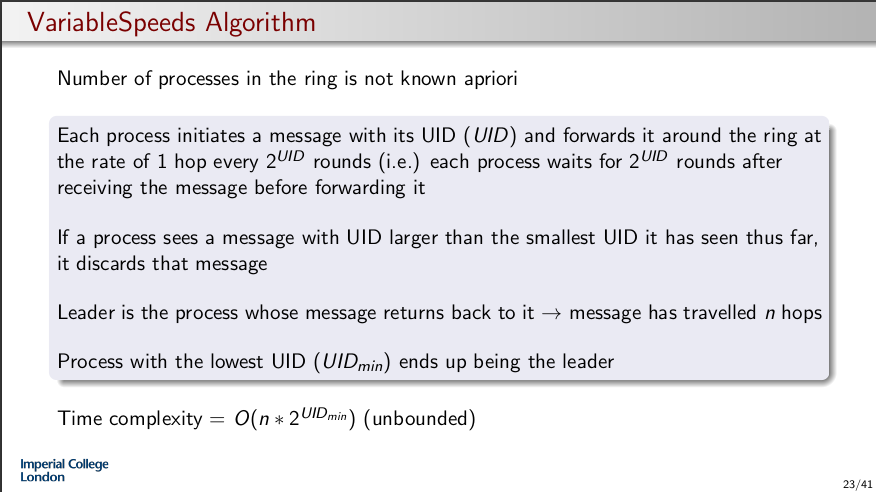
3a)

Conserves resources

Allows for non-symmetrical algorithms

b) Replace unifrom agreement with regular agreement

c)



Replace 2^UID with just UID, means that a UID is less likely to overtake another UID so the number of messages is higher.

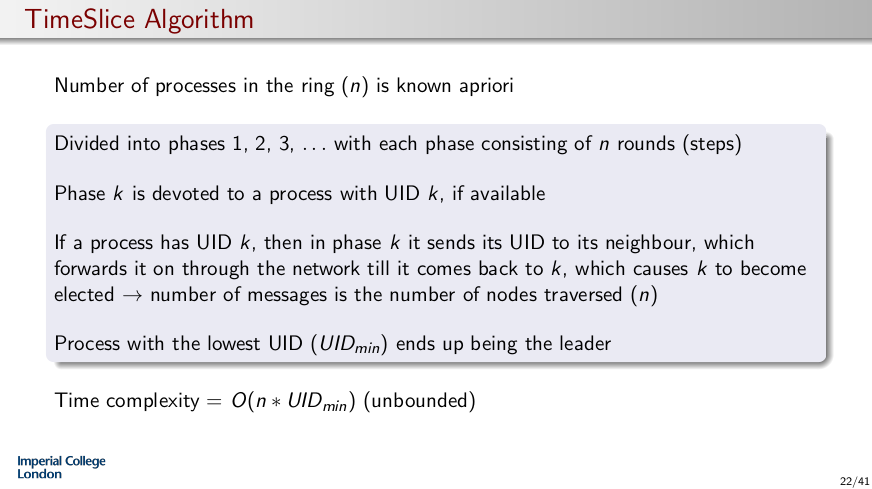
If we assume UIDs are positive integers, we need less time for the message form the process with smallest UID to complete a round

d) see page 313, cachin

e) Search may get stuck at one node as it’s async so it becomes DFS as there’s another branch that is not delayed that’s still searching while that one node is stuck.

Make it partially sync? Send pulses and only advance after receiving a pulse and a leaf node. If leaf node don’t forward pulse on else send pulse to children. Root waits for a pulse back from every node before sending another. In return of pulse send how many children you have so root knows how many it’s waiting on.

f)



Msq complexity is O(n)

See Anandha tutorial

4

See Anandha tutorial

a)

Synchronisation Variables:

Used to control access to the critical section as necessary

In a DSM, can be used to synchronise the memory → consistent view of the

Memory

b) Go to message central node then outer node then central then back to outer and so on.

Better answer:

If the lookup fails or crashes, the sender will not know at what point it crashed. With an iterative approach, iteratively taking steps at a time, if the lookup fails after being sent down a certain link, we can be more sure of where there might be a failure in the network.

c) Since the nodes all have O(1) links, easiest to search for a key would be to use depth first

search. Message complexity would be O(|E|) = O(N ), as each node only has O(1) links.

d) Causal consistency is good enough → main issue is the fact that we need to ensure that the

changes in stock values and the reaction to them are consistent. Stock value changes for independent

stocks can be seen in different order.

e) Easy to adjust for nodes joining/leaving the network

Arbitrary queries, such as keyword, range and attribute queries supported

Searching for a resource entails high message overhead and delays

Look-up isn’t O(1) performance

f) i) Yes, PO then P1 then P2. This order is obtained non-currently

ii) No, third read must happen after 2 writes. If reads a sync then the 2 vars must have been written to just before the second read so third print is 1,1. Same for writes

iii) Yes?

X=1

Print(y,z)

Y=1

Z=1

Print(x,z)

Print(x,y)