1(a) \*\*\*\*\*\*Code not tested\*\*\*\*\*

defp next c, pl, processes, alive, suspected, timeout do

receive do

{:timeout} ->

suspected = Enum.reduce(processes, suspected, fn p, acc ->

if !member?(alive, p) && !member?(suspected, p) do

send c, {:suspect, p}

MapSet.put(acc, p)

else

if member?(alive, p))&& member?(suspected, p) do

send c, {:restore, p}

MapSet.delete(acc, p)

end

end

end)

alive = MapSet.new

timeout = if MapSet.intersection(alive, suspected) == MapSet.new do

timeout \* 2

else

timeout

end

Process.sendafter self(), :timeout, timeout

next c, pl, processes, alive, suspected, timeout

{:pl\_deliver\_HBRequest, q} ->

send pl, {:pl\_send\_HBReply, q}

next c, pl, processes, alive, suspected, timeout

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

next c, pl, processes, MapSet.put(alive, p), suspected, timeout

end

end

1(b)

*Strong completeness:* Process that crashed will not be able to send a heartbeat reply, and therefore will never be in the alive Set. It will be added to the suspected Set during the first timeout after the crash.

*Eventual strong accuracy:* The timeout will be doubled every time. All the correct process should eventually have enough time to send a heartbeat reply and therefore be removed from the suspected Set if it was added into the Set before.

The initial timeout of EPFD can be set more aggressively so it reacts faster to failure. This may lead to false suspicions but EPFD permits it, while PFD does not. (refer to section 2.6.4 of Cachin)

The timeout in EPFD is closer to the network situation at each node, while PFD has to use the upperbound time t, so eventually EDFP is able to report crashed nodes faster.

2

a

i

safety

ii

~~Liveness~~  safety

iii

liveness

b

Nothing, since majority-ack uniform broadcast never assumes the usage of the agreement property.

===

Kinda agree with above, but if we have processes A,B,C:

A BEB some msg M, crashes after B delivers before sending to C, then M not URB delivered.

A RB msg M, crashes after B delivers but before sending to C, then B rebroadcasts, C delivers, and M URB delivered to B and C

I guess this kinda counts as a gain?

Basically you lose the addtional liveness property provided by RB, and that has some effects.

c

No.

The total order property says that “if a process (correct or crashed) delivers message M1 without previously delivering message M2, then no correct process delivers M2 before M1”.

It is not possible to guarantee the total-order property under BEB due to the fact that BEB does not check for acknowledgement before beginning the next broadcast. This means that we can have situations in which a process sends M1 then M2 out but M1 gets delivered after M2 due to network lag / link failure.

d

Should be similar to Lemma 2. Don’t have a concrete example at the moment.

All bits are [1, 1, 1, 0, 0, 0]

A1 recieves 1,1,1,0,0

A2 recieves 1,1,0,0,0

?

e

Without loss of generality we can assume that P has the larger number and Q has the smaller number.

|  |  |  |
| --- | --- | --- |
| The value of P | The value of Q | Result |
| 2 | 1 | The process Q will become the leader immediately since it knows that P must have 2 since it can’t have 0. |
| 3 | 2 | The process Q will become the leader after one second becomes it know that it can’t be in the previous situation thus the value of P is 3. |
| 4 | 3 | The process Q will become the leader after two seconds since it knows that it won’t be in the previous situation so P must have 4. |
| 5 | 4 | ... |

So, in general, the process with smaller value will become the leader after (v - 1) seconds.

Some other thoughts, the above need to be sure that both processes agree on the timestamp value and messages are delivered to both processes within 1 seconds. And their value can be arbitarily large so the actual time complexity is unbound. I propose an idea which does not need the above assumptions.

since P and Q are consective, one is odd and the other is even. Only process with odd value can propose a number to the other process. So after 2 time ticks, that process is guaranteed to become the leader.

3a)

If two processes install new views V = (id, M), and

V 0 = (id, M 0 ) respectively, then M = M 0

Follows directly from the uniform agreement property of the underlying uniform

consensus abstraction

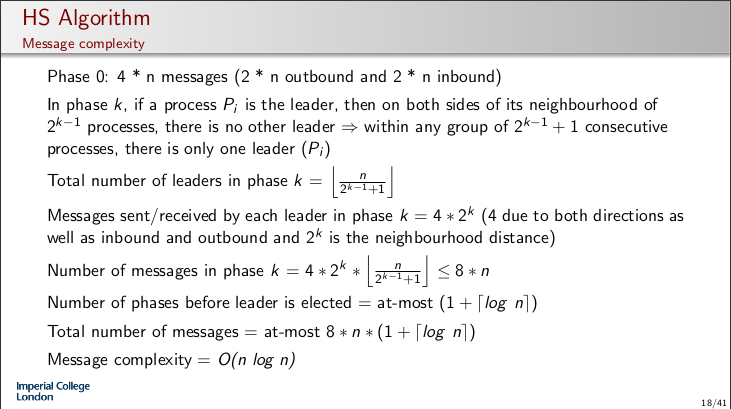
b)

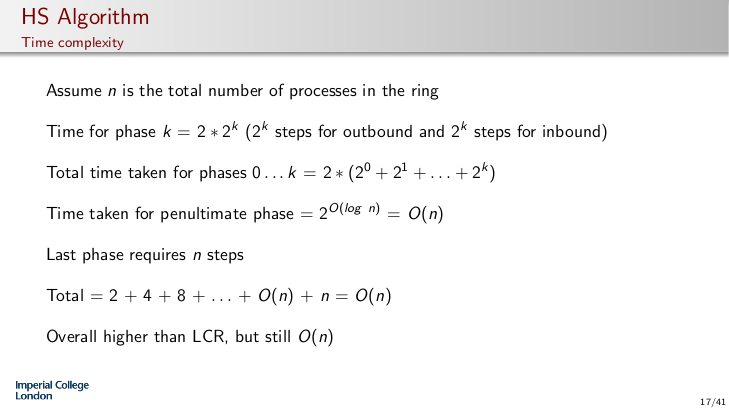
Safety ⇒ At-most one process is elected as the leader

Liveness ⇒ Eventually all processes should know a leader has been elected and at-least

one process is a leader

c)





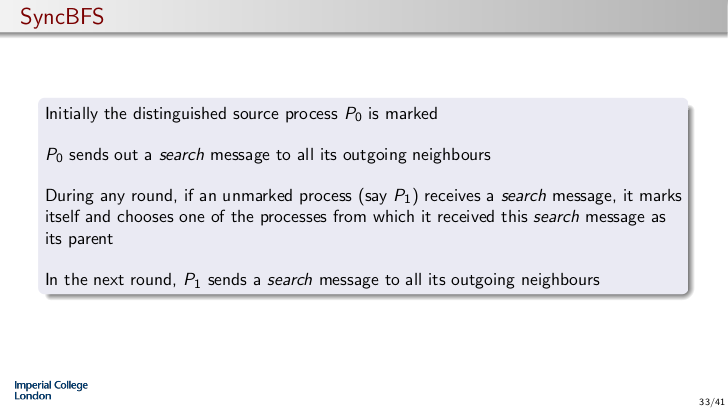
d)

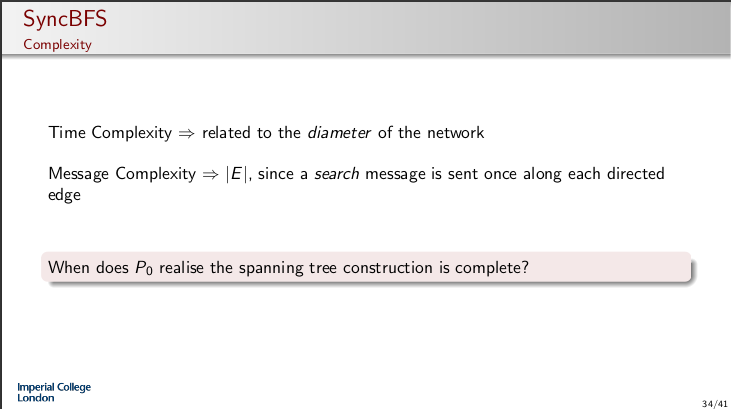
If a process p1 deliveries a message to another process p2 but then crashes, it means that that the view could be updated to not include p1 but then p2 deliveries p1’s message so other processes recieve messages from dead processes.

When wanting to install a new view block to give a chance for messages already sent to filter through.

e)

For election every node does SyncBFS and they search for UID\_min or UID\_max





4.a)

Programmers need to understand consistency models to write correct programs

Due to the DSM using asynchronous primitives, cannot be more efficient than

asynchronous message passing implementations

b)

No, only update the finger tables of nodes before/after the new node added

c)

Guarantees that any process can complete any synchronisation operation in a finite

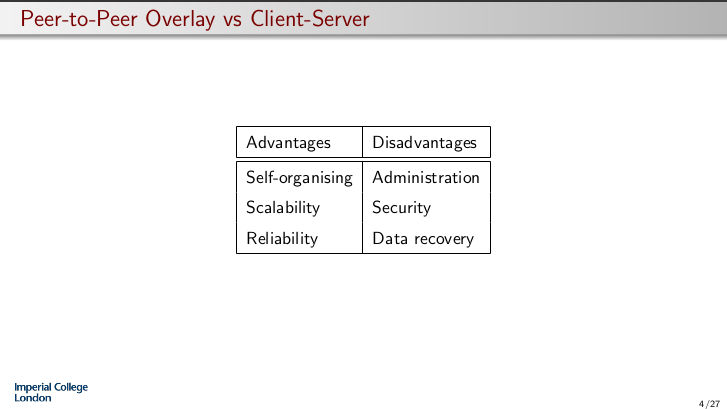
number of lower-level steps irrespective of the execution speed of other processes

Eg. producer-consumer:

If the producer process crashes without adding to the queue, the consumer would

be blocked

d)



e)

When joining broadcast meta-data about your files and then other nodes in the system can reply if they have files similar to that. Then when searching try the nodes that replied first before any other nodes. Bad for high churn though.

f

I think this should be the answer. It can’t be strict nor sequential. It is not causal as W(x,2) causally precedes W(x,5) because of the R(x,2) before it. Processor consistency applies, as W(x,2) and W(x,5) can arrive differently. Any thoughts?

Strict consistency - No

Sequential consistency - No

Causal consistency - No

Processor consistency - Yes

Slow-memory consistency - Yes