Welcome for discussion :)

1a(i).

Assumption:

1. The process id of each process equals to their self() value in Elixir.

2. Each round (successfully sending out one :urbc\_deliver) is independent to each other.

defmodule URBC do

def start(c, beb, pfd, uc) do

receive do

{:bind, processes, processB, c, beb, pfd, uc} -> next(processes, processB, c, beb, pfd, uc, false)

end

end

def next(processes, processB, c, beb, pfd, uc, crashed) do

receive do

{:pfd, pfd, crashedP} ->

if crashedP == processB do

send(uc, {:uc\_propose, {}, true})

next(processes, processB, c, beb, pfd, uc, true)

else

next(processes, processB, c, beb, pfd, uc, false)

end

{:urbc\_broadcast, c, msg} ->

# Only B broadcasts

if self() == processB do

send(beb, {:beb\_broadcast, msg})

next(processes, processB, c, beb, pfd, uc, crashed)

else

next(processes, processB, c, beb, pfd, uc, crashed)

end

{:urbc\_deliver, p, msg} -> x

if p == processB do

send(uc, {:uc\_propose, msg, false})

next(processes, processB, c, beb, pfd, uc, false)

else

next(processes, processB, c, beb, pfd, uc, crashed)

end

{:uc\_decide, msg, crashed} ->

if crashed do

send(c, {:urbc\_deliver, :crashed})

next(processes, processB, c, beb, pfd, uc, false)

else

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

next(processes, processB, c, beb, pfd, uc, false)

end

end

end

End

defmodule URBC do

def start do

receive do

{:start, c, processes, broadcastP, beb, uc} ->

next c, processes, broadcastP, beb, uc

end

end

defp next c, processes, broadcastP, beb, uc do

receive do

{:urbc\_broadcast, msg} when self() == broadcastP ->

send beb {:beb\_broadcast, msg}

{:beb\_deliver, msg} ->

send uc, {:uc\_proprose, msg}

{:pfd\_crash, process} ->

if process == broadcastP do

send uc, {:uc\_propose, :crashed}

end

{:uc\_decide, result} ->

send c, {:urbc\_deliver, result}

end

next c, processes, broadcastP, beb, uc

end

end

(ii).

Integrity: In the given implementation, :urbc\_deliver is only sent when receiving a decision from the consensus module, whose instance was only started after receiving either a message of or the crashing message of B. While from the “no creation” property that BEB inherits from PL and the “strong accuracy” property from PFD, such a message should only be delivered under the specified conditions.

Validity: By the “validity” property of BEB directly, if process B is correct, all correct processes it broadcasts to will deliver the broadcasted msg.

Uniform agreement: By the “uniform agreement” property of uniform consensus, process excepting B would either decide to deliver the crashed msg of B or the msg that B broadcasts. In case that any single process delivers M, then all others would do the same.

Termination: Since B broadcasts a message M, either that would be successful or B would crash. In either case other correct process will respond (propose msg or :crashed). So an instance of consensus will be started. Recall the “termination” property of UC, all correct processes will decide on either msg to send, in which cases they all send one message.

b. (not sure)

The properties of consensus are not satisfied. Since the selection on the number R is not specified, it can be chosen to be 1. Given that F processes can crash in such a system, there could be a cast that the crashed process proposes the minimum value in the current round, which is only received by a subset of other correct process at the very end of the round. Other processes would not have the knowledge of that minimun value and would choose a local minima. In this case, they decide on different values.

Since processes receive N-F values and ignore the other F values, it’s possible that the process with the minimum value gets ignored by all other processes. This inconsistent state can continue to any round after. Therefore, that0-

process cannot reach consensus with all other processes.

1c. (not a clean approach)

In each round, every process proposes values and sends them associated with their UID to the current leader, who decides which value to take regarding majority if possible or random selection otherwise. If the sender of a specific value is known to has crashed, its value would not be selected. The leader broadcasts the selected value back to other processes. Termination of the current round is either by receiving the decision made by the leader or detection of leader failure. In the next round, each process proposes the values that they receive from the former leaders or themselves if not available. Finally, the minimum value after the last round is decided. The final value is selected within the ones by leaders in some rounds.

For validity, the values that have ever been proposed are all proposed by some process.

For integrity, there is only one minimum value, or duplicate but still the same value.

For termination, each round is bound by either leader finishing broadcating or timeout.

For uniform agreement, there is at least one round where leader does not crash, then they will have at least one common value, which will the the finally decided value.

2.a

The leaving of a process can be simply modelled by considering it as crashes. While the joining of a process can be seen as the reversing process of exluding it (i.e. including it) from the current membership view. Particularly, a new process will broadcast its existance to all existing nodes. The nodes that receives it will start proposing a new view including the new node using consensus. Finally, they all install a view including the new node.

b. (not sure)

We cannot do any better than that, because it has been proved as a theorem that under a n-size ring setting, comparison-based leader election algortihms has the message complexity of \Omega(n\*log(n)).

c.

Use hash function such as SHA256 and feed the node’s properties including IP, hardware parameters plus some randomness like current timestamp to get the hash value as their UID. They rarely conflict for a good hash function. Extra phases are executed to find any duplicate, if unlikely there is, rehash the existing value.

(but I cannot think of how to perform the duplicate elimination)

d.

It is not quite true. The distributed memory systems could have been designed in a way such that strong consistency model is adopted. However, applying such a strict model blindly could lead to inefficiency of some application built upon them. Some tasks are less demanding on consistency properties but more on performance (e.g. small delay and message overhead). Or they might require a very fine granularity control of the underlying message passing primitives. For applications that do require a strong consistency model, they are normally encapsolated (from weaker models) and can be used as API (e.g. the C++ atomic library).

e. (not sure)

Assumption:

1. Finger tables are in used to shorten the query path

2. External processes that query the DHT system are relatively fixed and not malicious.

3. The nodes in the DHT system are not joining/leaving very frequently

Each time when a query is successful, the client stores the node id associated with the the file. On next request, it sends its knowledge on the node-file mapping as well. The nodes that process the query build up a collection of such mapping and might use it to perfrom direct forwarding later. After the system stablising, most queries would be forwarded directly to the correct node. The traffic routing will therefore be more regular. However, on node leaving, such mapping tables need to be maintained, which adds further overheads to node adjustment of a DHT system.

f.

Strict consistency is not satisfied, since P1 has written x as 2 then 4 right at the start, P3 still reads it to be 2 later, which voilates the global time ordering.

Sequential consistency is not satisfied, since P2 reads x as 4 then 7, while P4 does the other way around, they do not see the same ordering of the two writes issued by P1.

Casual consistency is not satisfied, P2 reads x as 4 then writes it as 7, which forms a causual relation, where the value 4 of x must appear before its value 7. But P4 reads x in a reversing order.

Processor consistency is satisfied, since only P1 doing two writes that form a restriction to the ordering, where other processes should read the value 4 after 2. However, no other process reads the two values in the given execution. So such a restriction is not voilated.

Slow memory consistency is satisfied, inherited from processor consistency. Specifically, the same restriction (in this case) as the processor consistency is required here, since they all write to x. The restriction is not voilated.