1 2001 Paper 9 Question 1

Breaks serializability

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2001p9q1.pdf

physical vibrations break the crystals -- voltage/temperature is invertible

(a) Explain the problem of clock drift in distributed systems.

vector clocks are LARGE

vector clocks struggle when nodes can leave and join the network

The clock in nodes in distributed systems are not perfect. Some run slightly faster and some slightly slower. The rate by which a clock runs fast or slow is its drift. After a long time, clocks can become out-of-sync and be no longer be monotonic – we can send a message and from the perspective of the recipient, it can have been from the future. — true, but why is that a problem?

If you have 4 bytes and 1k nodes then you have 4KB of overhead per message!

vector clocks introduce new failure modes

(b) What sources of conventional earth time might be used by computer systems? How would you estimate bounds on the accuracy of time received from such a source?

clients on satellites are receive-only!

NTP != geostationary satellite method

Conventional earth time can be determined from GPS. Geostationary satellites have caesium-133 atomic clocks. These geostationary satellites orbit the planet and broadcast the current time and their position. A server can ~~then~~ hear the signal from multiple satellites and using the delay between them, work out its own position and the current time. You can estimate bounds on the accuracy by repeating the process multiple times or repeating with different satellites.

We can correct systematic errors by changing assumptions, measuring voltage, temperature, atmospheric pressure etc etc

Correct random error by getting multiple samples

exemplifying what principle(s)?

NTP assumes delay is symmetric

Repeat NTP with different clocks!

If you have a clock and someone else has a more accurate clock then almost all of the error is from your own clock -- so you ask them for their time and believe them

On an atomic clock, ask other atomic clocks and average

(c) What constraint does distributed inter-process communication (IPC) impose on the clock values of the communicating parties?

Most networks assume delay is Poisson -- but it's usually not due to bursts

Distributed inter-process communication requires that each parties clock respects causality. The timestamp of any send must be earlier than the timestamp of the corresponding receive. This is difficult because the timestamps are generated by different clocks.

Build an emprical distribution for delay across the network and find a MLE for the time

(d) Outline one clock synchronisation protocol that satisfies this constraint.

Vector clocks respect causality. A vector clock keeps track of the last operation each node did that could have affected the current process. Vector clocks do not use physical time, but logical time.

A vector clock timestamp in a system with n nodes is a n-dimensional vector where the n^{th} dimension represents the last known operation at node n.

If Node i has vector timestamp t then here are the protocols for operations:

- To send a message, increment $t[i]$ then send the message along with the new timestamp
- On receiving a message with timestamp t', take the elementwise maximum of t' and t. Let this be t''. Increment $t''[i]$ and let t'' be the new timestamp.

If we have two timestamps, T_1 and T_2 then $T_1 < T_2$ if and only if $(\forall i. T_1[i] \leq T_2[i]) \wedge (\exists i. T_1[i] < T_2[i])$.

If we have two timestamps T_1 and T_2 such that $T_1 \not< T_2 \wedge T_2 \not< T_1$ then we say that T_1 and T_2 are concurrent, denoted $T_1 \| T_2$.

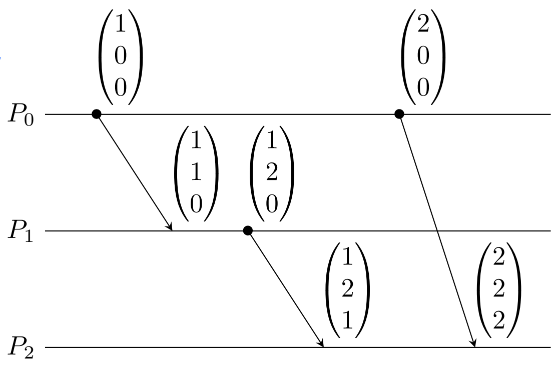
ish. n=2 here, for IPC!

All this simplifies down to Lamport Logical Clocks and T_{rx}=1+max(local,incoming)

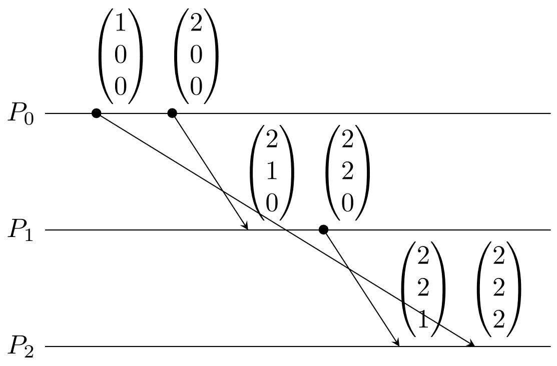
(e) For each of the cases of IPC illustrated below, give the vector clock values that message receiving modules and delivery modules could maintain for each process.

Label events as they happen so in retrospect you can answer this

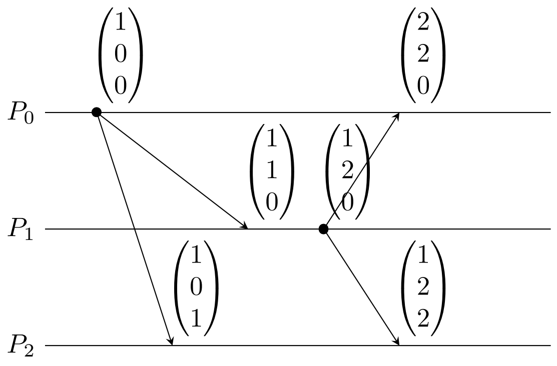
The timestamp increases by 1 on every send or receive



(i) ✓



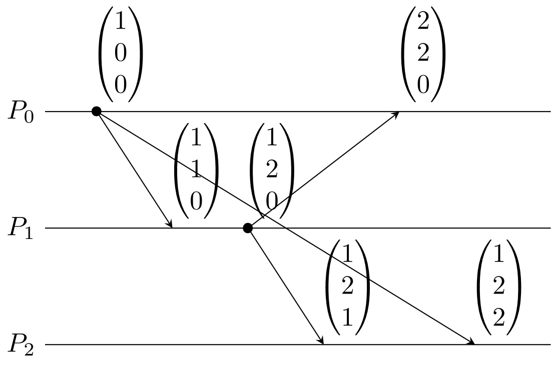
(ii) ✓



(iii) ✓

Causal order: a → b

Is a partial order
Cannot generate a total order



(iv) ✓

(f) Define "causal order" of message delivery. In which, if any, of (*i*) to (*iv*) above is causal order violated at the message receiving module?

Happens before:
a < b if a and b are on the same machine and happened in that order
a < b if a is the send and b is the receive
closed under transitivity

To find out if causality has been violated:
start with "happens-before"
If you receive a message which "happened before" your current timestamp then causality was violated.

Causal order of message delivery means that the order of message delivery cannot violate the "happens-before" relation of the times the messages were sent. ✓

Informally this means that a message *a* sent to node *n* cannot be delivered until all messages sent to node *n* which were logically sent before *a* have been delivered. ✓

Using vector clocks, causal order of message delivery is violated if a message with timestamp t' is delivered to a node with timestamp t such that $\forall i. t'[i] < t[i]$. yes but how is < defined?

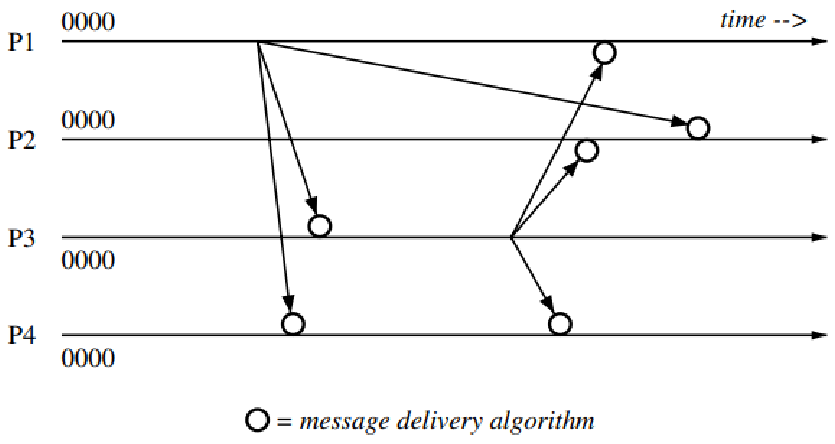
✗ Only *ii* violates causal order of message delivery. and (iv)

✓ In *ii*, the message sent from P_0 to P_2 logically happens before the message sent from P_0 to P_1. Therefore by transitivity of the happens-before relation, it logically happens before the message from P_1 to P_2. However, the message from P_0 to P_2 was delivered after the message from P_1 to P_2. Therefore it was delivered out-of-order.

✗ Note that *iv* does not violate causal order of message delivery since the first and second message happened concurrently. Incorrect... They did not happen concurrently: the delivery of message 1 caused the delivery of message 2

2 2005 Paper 9 Question 4

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2005p9q4.pdf



Causal Broadcast HAS TO SEND to every node on the network

You can't send to x people on the network where x < n

Can include metadata to "disregard this if you're not in set s"

v[i] is the number of messages received from node i

Any node which receives the message must have at least as many

The above diagram represents a process group that communicates by means of multicast messages. At each process-hosting node, message delivery software decides whether a given incoming message should be delivered to the process or buffered for later delivery. This is achieved by the use of vector clocks.

Increment AFTER sending -- not before
simpler to write code
Your own message can be delivered immediately

OR you can send it to yourself.
But messages CANNOT have side effects.

can implement TOB on top

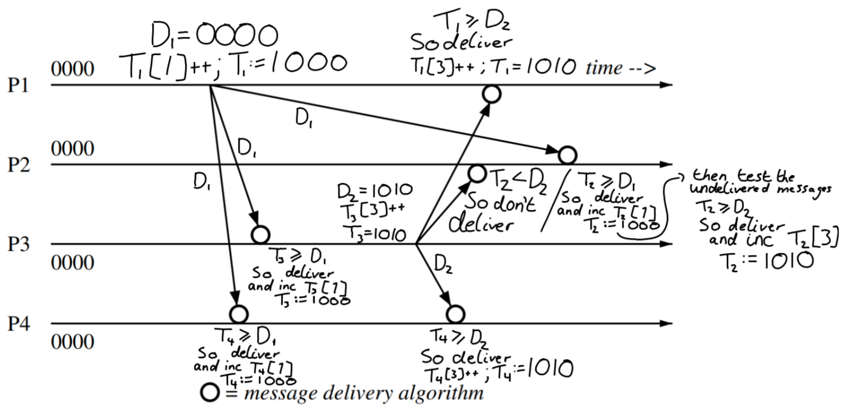
(a) Describe, by means of the above example, the vector clock algorithm for delivery of messages in causal order.

The vector clock algorithm can be used to ensure broadcast messages are delivered in causal order. ✓

Each node i has a vector $\mathbf{v}_i$ of size n where n is the number of processes in the system. The vector clock algorithm maintains the invariant that $\mathbf{v}_i[j]$ is the number of messages sent from node j which node i has received. ✓

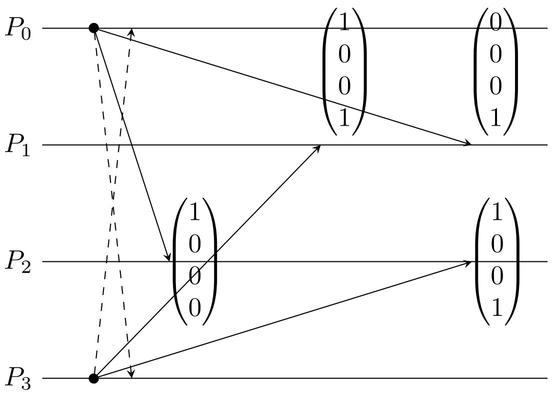
When node i wishes to broadcast a message, it includes its node id (i) and a copy of $\mathbf{v}_i$ in the message and *then* increments $\mathbf{v}_i[i]$. The vector timestamp shall be used as a list of "dependencies" before the message can be delivered. ✓

When node i receives a message, it adds it to a set of waiting messages. Then node i checks for any messages with a dependency list that is satisfied by its current timestamp. If it finds any, it delivers them and increments $\mathbf{v}_i[j]$ to maintain the invariant that $\mathbf{v}_i[j]$ is the number of messages delivered to process i which were sent by process j.



(b) By means of a similar example, show that total ordering of messages is not achieved by this algorithm. Total order Broadcast is required to resolve conflicting operations (operations which don't commute)

Consider the below example.



P_0
P_1
P_2
P_3
$\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$
$\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$
$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}$
$\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$

In this example, P_0 and P_3 send messages concurrently and therefore any order of delivery will respect causal order. In this case, processes P_1 and P_2 deliver in two different orders which both respect causal order. This demonstrates that the vector clock algorithm does not enforce a total order. Note that to avoid clutter, I have only written the most important timestamps.

3 2004 Paper 8 Question 4

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2004p8q4.pdf

(a) Define strong and weak consistency.

Strong consistency (linearizability) means that every operation takes place atomically across all replicas at some point after it started and before it finished.

Weak consistency (serialisability) means that operations have the same effect as if they were executed in some serial order.

Informally, weak consistency respects only the happens-before relation, while strong consistency also respects real-time.

(b) A process group manages a set of widely distributed replicas. The group is open and unstructured; that is, external processes may invoke any group member for reading or writing.

(i) Discuss how the replicas can be kept strongly consistent in the presence of concurrent invocations and failure.

If using a commit manager then they write onto disk -- so on recovery, you can see the result of the commit

The ABD algorithm supports gets and blind sets with strong consistency and tolerates of up to half of the nodes on the network failing.

Writes to out-of-date updates can update to the MOST RECENT copy without need to iterate through all old copies

The ABD algorithm is described as follows:

- Get()
  - Get a quorum of nodes to give a value
  - Read-repair all nodes with an outdated value ✓
  - Deliver the value *after* read-repairing

So using quorums doesn't delay work -- it decreases it

Each variable needs a version number

- Set()
  - Get the logical timestamp from a quorum of nodes (including read-repairing) ✓
  - Send the set operation to any number of nodes. It will be committed when a quorum of nodes has seen it.

Persist to disk THEN send messages

2 phase commit?

Under the described algorithm, the node invoking the get/set does a linear amount of work for each read and write. This means that the overall bandwidth of the network is high, but the work done by the caller and the latency is also very high. ✓

We can decrease the work the caller has to do by using a gossip protocol to distribute writes (write to k random nodes) and reads (ie read from k nodes you know exist and return the results).

Using a Gossip protocol, each node has to do constant work for each read and write. The work required for a write on each node is now asymtotically optimal – so many write requests can be made across the whole system very quickly without impacting any single node. However, reads still require a linear amount of work: the network still has very high latency on reads. ✓

Under this algorithm, the rate at which requests can be served is $\approx 2n$ where n is the number of requests a node on the system can process.

Alternatively, we could use an algorithm like Google Spanner to implement reads with a constant amount of work. However, the question was written before the Spanner paper was published; so I am confident this is not the expected answer.

:D That depends whether you think Spanner was a new idea or the first implementation of an idea.

(ii) Would it be more appropriate to use a structured group (with a single coordinator) to manage the replicas? Justify your answer.

It would not be more appropriate to use a structured group. In such a group, every read or write request would have to go through the leader. So the number of requests the network can process is equal to the number of requests the leader can process – half that of the other solution.

No, can't ulilaterally declare that for all use cases.

depends on requirements. What's proposed is DNS, right, and that seems to work…

DNS does not support strong consistency -- only weak consistency

We can use the election algorithm from RAFT to ensure there is exactly one leader, make the network failure-tolerant and implement FIFO-total order broadcast to implement serializability. However, a structured group cannot efficiently implement linearizability. If we use a leader, then all writes must go through the leader (obviously). However, all reads must also go through the leader – else we could read an out-of-date copy and break linearizability. ✓

The two solutions to this are:

- Allow arbitrary delays on reads

  This solution is unworkable – reads could be blocked indefinitely.

- Forward the read onto the leader.

  This is unworkable – the leader now has to process every request on the network. This means the speed of the network is limited by the speed of a single node – the network is now equivalent to performing reads and writes on a single node with significant overhead from forwarding messages.

The stress on the leader could be alleviated by having multiple leaders for different sections of the database. /

Provide fast read and consistent reads as two operations?

Your solution should discuss the *selection and use* of algorithms and protocols. It is not necessary to specify them in great detail.

This is asking for DNS -- which is a tree structure. Different stratum. If you want an "authoritative answer" then you ask the root.
If you want a "non-authoritative" answer then you read from a lower node.

DNS can be modified to have a small quorum (ie 5) at the root instead of a single node

The main bottleneck is typically disk bandwidth

So minimising number of writes increases throughput