Distributed Systems: Solution notes for exercises

Martin Kleppmann (martin.kleppmann@in.tum.de)

Exercise 1. *A TCP connection allows two nodes to send each other arbitrarily long sequences of bytes. You decide that you want to send multiple requests and responses over the same TCP connection. What do you need to do in order to implement such a request-response protocol using TCP?*

Once you have encoded a request or response message as a sequence of bytes, you need some way for the sender of a message to tell the recipient where one request/response ends and the next one begins. This is known as *message framing*. One way is for each message to contain a header indicating the size of the following message in bytes. Another approach is to reserve a certain sequence of bytes as message separator, and ensure that this sequence cannot occur in the middle of a message (for example, HTTP/1.1 uses `\r\n\r\n` as message separator for requests).

Next, to associate requests and responses, one option is to rely on ordering: the first response sent by one node corresponds to the first request sent by the other node, and so on for the second, third, etc. request/response (HTTP/1.1 uses this approach). A downside of this approach is that if one request is slow to process, all of the following requests must wait until the slow one is complete before their responses can be sent (this is known as *head-of-line blocking*). An alternative is to give each request a unique ID of some sort, and for each response to quote the ID of the request it is responding to. That allows the responses to be sent in a different order from the requests, which means a response can be sent as soon as it is ready, without waiting for other requests (HTTP/2 works like this).

Exercise 2. *How is RPC different from a local function call? Is location transparency achievable?*

- RPCs may time out if the request or response message is lost. Even if we retry, there is no guarantee that the messages will get through. The application must handle this error condition, and the possibility of failure may need to be reflected in the type signature. For example, Java RPC libraries often throw a checked exception, and JavaScript RPC clients often return a *promise*, which can either succeed or fail.
- If a timeout occurs, the RPC client doesn't know whether the server executed the function; local function calls don't have this uncertainty.
- RPC is often much slower than a local function call, due to network latency. Moreover, network latency is often variable and unpredictable, while the execution speed of a local function is usually predictable.
- RPC clients and servers may need to take measures to make function invocation *idempotent* (this concept is introduced in Lecture 5), to allow safe retries in case of message loss.
- Another difference is that if an object reference is passed as an argument, a local function call can potentially mutate the object (unless it is immutable). This is not possible with RPC, since the remote function cannot access the caller's memory; the only way how the remote function can pass information back to the caller is through the return value, or through another RPC.
- Perfect location transparency can only be achieved by making local function calls more like RPCs (e.g. allowing them to unpredictably fail). This is undesirable in most cases, and therefore most systems do not attempt to completely hide the distinction between local and remote calls.

Exercise 3. *Say you have a client-server RPC system in which a client repeats an RPC request until it receives a response. How could the server deduplicate the requests?*

When the server receives several identical requests, it needs to somehow determine whether they are duplicates, or whether the client really wanted the action to take place several times. One solution is for the client to attach a unique identifier to each distinct request, and to include the same identifier in all retries of that request. The server can then keep track of all request identifiers it has processed, and ignore any requests with identifiers it has already seen. This approach has the effect of making the RPC *idempotent*, which we will discuss further in Lecture 5.

Exercise 4. *Reliable network links allow messages to be reordered. Give pseudocode for an algorithm that strengthens the properties of a reliable point-to-point link such that messages are received in the order they were sent (this is called a FIFO link), assuming an asynchronous crash-stop system model.*

The sender attaches a sequence number to each message, and increments that number for each message sent. The recipient delivers messages in increasing order by sequence number. This may require buffering a message on the recipient side until messages with prior sequence numbers have arrived.

```
on initialisation do
    [We have a separate copy of the following variables for each network link.]
    $lastSent := 0;\ nextReceived := 1;\ buffer := \{\}$
end on

on request to send message $m$ over FIFO link do
    $lastSent$++
    send $(m, lastSent)$ via the underlying network link
end on

on receiving $(m, n)$ via the underlying network link do
    $buffer := buffer \cup \{(m, n)\}$
    while $\exists m'.\ (m', nextReceived) \in buffer$ do
        deliver $m'$ to the application
        $nextReceived$++
    end while
end on
```

Exercise 5. *How do we need to change the algorithm from Exercise 4 if we assume a crash-recovery model instead of a crash-stop model?*

When a node crashes and recovers, all of its in-memory state is lost. The sequence number algorithm maintains state in the form of the variables *lastSent*, *nextReceived*, and *buffer*. This information needs to be preserved across crashes, so it must be maintained in non-volatile storage (e.g. on disk).

Another issue is that a node may crash while in the process of sending or delivering a message. In that case, we will need to decide: does that message get sent/delivered after recovering from the crash? If so, the information about such in-progress operations will also need to be written to non-volatile storage.

Exercise 6. *Describe some problems that may arise from leap second smearing.*

Measurements of time duration will be slightly wrong during the smearing period. For example, if the extra second is spread over 24 hours (12 hours before and 12 hours after the leap second), this amounts to an error of about 12 ppm in the rate at which clocks are running. However, many systems can tolerate this level of inaccuracy.

A bigger problem is that if a timestamp from a smeared clock is compared with a non-smeared timestamp, there will be a spurious difference of up to half a second. For example, this could arise if two nodes with different smearing policies attempt to measure the network latency between them by sending one node's current timestamp as a message to the other node, and calculating the difference of timestamps when the message is received. The measured latency could even be negative.

A similar problem arises (to a lesser degree) when comparing two timestamps from clocks that use different approaches to smearing. For example, some implementations of smearing spread the extra second over different periods of time. Also, different functions are used to interpolate across the leap second discontinuity: some systems use a linear functions, other use a cosine. There is no standardised approach to smearing. See https://developers.google.com/time/smear for some discussion.

Exercise 7. *What is the maximum possible error in the NTP client's estimate of skew with regard to one particular server, assuming that both nodes correctly follow the protocol?*

The maximum possible error in θ is $\delta/2$. This occurs either in the extreme case where request latency is 0 and response latency is δ, or in the other extreme case where request latency is δ and response latency is 0. Whenever latency is nonzero, the error will be less than $\delta/2$.

Exercise 8. *A relation R is a* strict partial order *if it is irreflexive ($\nexists a.\ (a, a) \in R$) and transitive ($\forall a, b, c.\ (a, b) \in R \wedge (b, c) \in R \implies (a, c) \in R$). (These two conditions also imply that it R asymmetric, i.e. that $\forall a, b.\ (a, b) \in R \implies (b, a) \notin R$.) Prove that the happens-before relation is a strict partial order. You may assume that any two nodes are a nonzero distance apart, as well as the physical principle that information cannot travel faster than the speed of light.*

The fact that happens-before is transitive follows directly from the third bullet point of its definition.

To prove that it is irreflexive, we use proof by contradiction. Assume to the contrary that there exists a such that $a \to a$. This implies at least one of the following:

1. a occurred before a in some node's local execution order. However, this contradicts our assumption that every node's local execution forms a strict total order, which is irreflexive.

2. a is the sending of some message m, and a is the receipt of that same message m. This is impossible because sending and receiving a message are two different events.

3. There exists an event b such that $a \to b$ and $b \to a$. Let N_a be the node at which a occurs, and N_b the node at which b occurs. Then either $N_a = N_b$ or $N_a \neq N_b$. If $N_a = N_b$ then $a \to b$ and $b \to a$ contradict our assumption that every node's local execution forms a strict total order, so we have $N_a \neq N_b$.

   Let t_a be the time at which a occurs and let t_b be the time at which b occurs, both according to N_a's clock. Let d be the distance in space between N_a and N_b. $a \to b$ implies the existence of a message, or a sequence of messages (possibly via other nodes), through which information can travel from a to b. Since this information cannot travel faster than the speed of light, we have $t_b \geq t_a + \frac{d}{c}$, where c is the speed of light. By similar argument, $b \to a$ implies that $t_a \geq t_b + \frac{d}{c}$.

   Since we are assuming $d > 0$ and the speed of light is finite, $\frac{d}{c} > 0$. Therefore we have $t_b > t_a$ and $t_a > t_b$, a contradiction.

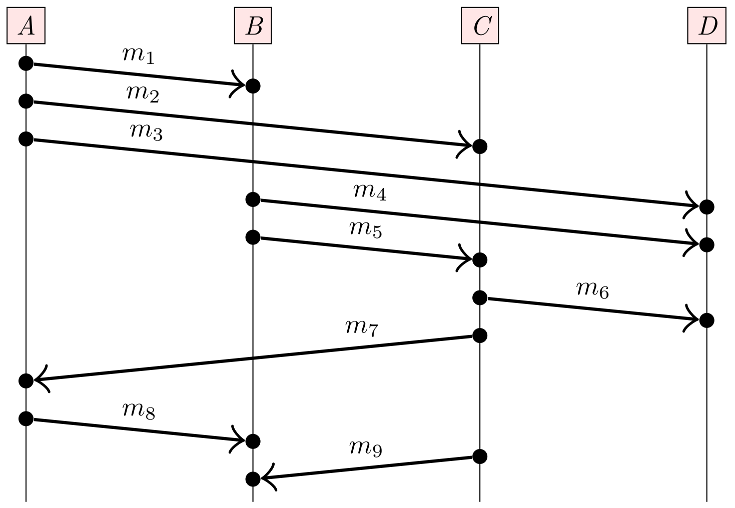
Exercise 9. *Show that for any two events a and b, exactly one of the three following statements must be true: either $a \to b$, or $b \to a$, or $a \parallel b$.*

The statement $a \to b$ must be either true or false. If it is true, then it is not the case that $a \parallel b$ by definition. Moreover, since the happens-before relation is a strict partial order (Exercise 8), $a \to b$ implies $b \not\to a$.

If $a \not\to b$, consider the statement $b \to a$, which must be either true or false. If it is true, then it is not the case that $a \parallel b$ by definition. If it is false, then $a \parallel b$ by definition.

In all cases examined, exactly one of the three statements $\{a \to b,\ b \to a,\ a \parallel b\}$ is true.

Exercise 10. *Given the sequence of messages in the following execution, show the Lamport timestamps at each send or receive event.*



The following table shows both the Lamport timestamps from this exercise and the vector timestamps from Exercise 12.

Message	Lamport				Vector			
	A	B	C	D	A	B	C	D
initial	0	0	0	0	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,0\rangle$
m_1	1	2	–	–	$\langle 1,0,0,0\rangle$	$\langle 1,1,0,0\rangle$	–	–
m_2	2	–	3	–	$\langle 2,0,0,0\rangle$	–	$\langle 2,0,1,0\rangle$	–
m_3	3	–	–	4	$\langle 3,0,0,0\rangle$	–	–	$\langle 3,0,0,1\rangle$
m_4	–	3	–	5	–	$\langle 1,2,0,0\rangle$	–	$\langle 3,2,0,2\rangle$
m_5	–	4	5	–	–	$\langle 1,3,0,0\rangle$	$\langle 2,3,2,0\rangle$	–
m_6	–	–	6	7	–	–	$\langle 2,3,3,0\rangle$	$\langle 3,3,3,3\rangle$
m_7	8	–	7	–	$\langle 4,3,4,0\rangle$	–	$\langle 2,3,4,0\rangle$	–
m_8	9	10	–	–	$\langle 5,3,4,0\rangle$	$\langle 5,4,4,0\rangle$	–	–
m_9	–	11	8	–	–	$\langle 5,5,5,0\rangle$	$\langle 2,3,5,0\rangle$	–
final	9	11	8	7	$\langle 5,3,4,0\rangle$	$\langle 5,5,5,0\rangle$	$\langle 2,3,5,0\rangle$	$\langle 3,3,3,3\rangle$

Exercise 11. *Prove that the total order $\prec$ using Lamport timestamps is a causal order.*

We assume $a \to b$ for some events a and b. Because the happens-before relation is a strict partial order (Exercise 8) we have $a \neq b$. To show that $a \prec b$ we use induction over the recursive structure of the definition of the happens-before relation. That is, we consider three cases:

1. First base case: $a \to b$ because a and b occurred at the same node, and a occurred before b in that node's local execution order. Then $L(a) < L(b)$ because the algorithm for Lamport timestamps only ever increases a node's local clock, and every local event increases the clock, so an event that occurs later in a node's local execution must have a greater timestamp. Hence $a \prec b$.

2. Second base case: $a \to b$ because $a = \mathsf{send}(m)$ and $b = \mathsf{receive}(m)$ for some message m. Then $L(a)$ is included in the message m, and so $L(b) = \max(t, L(a)) + 1 > L(a)$ by the definition of the Lamport timestamp algorithm. Therefore $L(a) < L(b)$ and hence $a \prec b$.

3. Inductive step: $a \to b$ because there exists an event c such that $a \to c$ and $c \to b$. Then $a \prec c$ and $c \prec b$ by the inductive hypothesis. Since $\prec$ is a total order, it is transitive, and therefore $a \prec b$.

Exercise 12. *Given the same sequence of messages as in Exercise 10, show the vector clocks at each send or receive event.*

See solution notes for Exercise 10.

Exercise 13. *Using the Lamport and vector timestamps calculated in Exercise 10 and 12, state whether or not the following events can be determined to have a happens-before relationship.*

Events		*Lamport*	*Vector*
$\mathsf{send}(m_2)$	$\mathsf{send}(m_3)$		
$\mathsf{send}(m_3)$	$\mathsf{send}(m_5)$		
$\mathsf{send}(m_5)$	$\mathsf{send}(m_9)$		

Note that although Lamport clocks cannot determine *happens-before* between nodes, they actually are sufficient to do so within a single node, as a higher local logical time will always reflect a later event.

Events		Lamport	Vector
$\mathsf{send}(m_2)$	$\mathsf{send}(m_3)$	$2 < 3$	$\langle 2,0,0,0\rangle < \langle 3,0,0,0\rangle$
$\mathsf{send}(m_3)$	$\mathsf{send}(m_5)$	cannot tell	$\langle 3,0,0,0\rangle \parallel \langle 1,3,0,0\rangle$
$\mathsf{send}(m_5)$	$\mathsf{send}(m_9)$	cannot tell	$\langle 1,3,0,0\rangle < \langle 2,3,5,0\rangle$

Exercise 14. *We have seen several types of physical clocks (time-of-day clocks with NTP, monotonic clocks) and logical clocks. For each of the following uses of time, explain which type of clock is the most appropriate: process scheduling; I/O; distributed filesystem consistency; cryptographic certificate validity; concurrent database updates.*

Process scheduling: If we are talking about the operating system scheduler on a single machine, we only need to measure elapsed time (e.g. how many milliseconds a particular thread has been running), so a monotonic clock is the most appropriate. However, if we are talking about tasks scheduled to run on a particular date at a particular time (e.g. using the Unix service `cron`), a synchronised time-of-day clock is required.

I/O: Timeouts and retry timers only need to measure elapsed time, so a monotonic clock is fine.

Distributed filesystem consistency: When tracking file changes, we need to ensure that changes are consistent with causality, which requires a logical clock. Probably a vector clock is most appropriate, because it would allow detecting when two different users concurrently modify the same file.

However, some software also relies on physical last-modification timestamps in a filesystem: for example, `make(1)` determines whether a file needs to be recompiled based on whether the source file's last-modification-time is earlier or later than that of the compilation output file. If the source file is written on one node but the compilation is performed on another node, this requires modification timestamps to be consistent with causality. It is also possible that a source file edit happens concurrently with a compilation, which could only be detected using vector clocks. However, last-modification timestamps are conventionally taken from a time-of-day clock, and changing these to be vector clocks would introduce compatibility problems.

Cryptographic certificate validity: A certificate includes a start date (when it becomes valid) and end date (when it ceases to be valid). If a node wants to check whether it is within that period, a synchronised time-of-day clock is required.

Concurrent database updates: Say two clients concurrently read the same object, modify it, and then write back the modified object to the database. The database will want to distinguish this from the situation where first client A reads and writes the object, and then client B reads and writes the object (where the version read by B includes A's changes). In the first case, the two updates are concurrent, while in the second case, B's update happened after A's update, so B's update should overwrite A's update. Distinguishing these two cases requires vector clocks.

An alternative approach is that the database does not detect whether updates are concurrent, but simply always lets an update with a greater timestamp overwrite an update with a lesser timestamp. In this case, Lamport clocks are best, since they provide a total order that is consistent with causality.

Exercise 15. *Prove that causal broadcast also satisfies the requirements of FIFO broadcast, and that FIFO-total order broadcast also satisfies the requirements of causal broadcast.*

To prove that causal broadcast is stronger than FIFO broadcast, assume an execution of causal broadcast containing at least two broadcast messages m_1 and m_2. If the two messages are broadcast by different nodes, FIFO broadcast does not require them to be delivered in any particular order, so we can ignore this case. If the two messages are broadcast by the same node, one of them must have been broadcast first. Without loss of generality, assume that m_1 was broadcast first and m_2 second. Then $\mathsf{broadcast}(m_1) \rightarrow \mathsf{broadcast}(m_2)$ by the first bullet point of the definition of the happens-before relation. Causal broadcast will therefore deliver m_1 before m_2 at every node, which meets the requirements of FIFO broadcast.

To prove that FIFO-total order broadcast is stronger than causal broadcast, assume an execution of FIFO-total order broadcast containing at least two broadcast messages m_1 and m_2. Per Exercise 9 we have either $\mathsf{broadcast}(m_1) \rightarrow \mathsf{broadcast}(m_2)$ or $\mathsf{broadcast}(m_2) \rightarrow \mathsf{broadcast}(m_1)$ or $\mathsf{broadcast}(m_1) \parallel \mathsf{broadcast}(m_2)$. In the case where the events are concurrent, causal broadcast does not require any particular message ordering, so we can ignore this case. This leaves the two cases where one broadcast happened before the other; we choose $\mathsf{broadcast}(m_1) \rightarrow \mathsf{broadcast}(m_2)$ without loss of generality. There are now two cases:

1. m_1 and m_2 were broadcast by the same node. Then $\mathsf{broadcast}(m_1) \rightarrow \mathsf{broadcast}(m_2)$ implies that this node broadcast m_1 before m_2. FIFO ordering requires that all nodes deliver these messages in the order they were sent, so all nodes deliver these two messages in causal order.

2. m_1 and m_2 were broadcast by different nodes. Then $\mathsf{broadcast}(m_1) \rightarrow \mathsf{broadcast}(m_2)$ implies the existence of a series of zero or more messages $M_1, M_2, \dots$ such that m_1 was delivered by the sender of M_1 before M_1 was broadcast, M_1 was delivered by the sender of M_2 before M_2 was broadcast, and so on, and M_k was delivered by the sender of m_2 before m_2 was broadcast.

   A message can only be delivered after it has been broadcast, and since total order broadcast requires all nodes to deliver messages in the same order, this implies that m_1 is delivered before M_1 on all nodes, M_1 is delivered before M_2 on all nodes, and so on, and M_k is delivered before m_2 on all nodes. By transitivity, m_1 is delivered before m_2 on all nodes, so the delivery order of these two messages matches their causal order.

Thus, FIFO-total order broadcast always delivers messages in causal order.

Exercise 16. *Give pseudocode for an algorithm that implements FIFO-total order broadcast using Lamport clocks. You may assume that each node has a unique ID, and that the set of all node IDs is known. Further assume that the underlying network provides reliable FIFO broadcast.*

Each node maintains its state in the following variables:

- `procs`: set of all node IDs in the group
- `proc`: ID of the current node
- `counter`: an integer, initially zero
- `minLamport`: a map where the keys are node IDs and values are integers (initial value 0 for each node)
- `holdback`: a priority queue where keys are (integer, node ID) pairs and values are messages; the `getMin()` method returns the entry with the smallest key according to lexicographic order on (integer, node ID) pairs

```
// Called by the user when they want to send a message
function totalOrderBroadcast(msg) {
  counter++;
  let m = (counter, proc, msg);
  sendFIFO(m); // use underlying FIFO broadcast
}

// Called by the FIFO broadcast layer when a message is received
function deliverFIFO(m) {
  let (msgCounter, sender, msg) = m;
  if (counter < msgCounter) counter = msgCounter; // Lamport clock update
  counter++;
  holdback.add((msgCounter, sender), msg);
  minLamport[sender] = msgCounter; // latest timestamp seen from each node
  tryDelivery();
}

// Examines the holdback queue. Delivers any messages that are ready to the
// application, in increasing Lamport timestamp order.
function tryDelivery() {
  let threshold = getThreshold();
  while (!holdback.empty()) {
    let (timestamp, msg) = holdback.getMin();
    if (timestamp > threshold) break;
    deliverTotalOrder(msg);
    holdback.remove(timestamp);
  }
}

// Returns the threshold, which is the minimum latest timestamp across all
// nodes. Due to FIFO ordering we know that all future messages delivered
// to deliverFIFO() will have a timestamp greater than this threshold.
function getThreshold() {
  let minimum = (+infinity, null);
  foreach p in procs {
    let timestamp = (minLamport[p], p);
    // The following comparison uses the total order on Lamport timestamps
    if (timestamp < minimum) minimum = timestamp;
  }
  return minimum;
}
```

Exercise 17. *Apache Cassandra, a widely-used distributed database, uses a replication approach similar to the one described here. However, it uses physical timestamps instead of logical timestamps, as discussed here:* https://www.datastax.com/blog/why-cassandra-doesnt-need-vector-clocks. *Write a critique of this blog post. What do you think of its arguments and why? What facts are missing from it? What recommendation would you make to someone considering using Cassandra?*

The article conflates several different things. It explains the advantage of performing fine-grained updates (e.g. updating only one field of a user record rather than replacing the whole record), which may reduce the frequency of conflicts between writes and may improve performance. However, the granularity of updates has got nothing to do with the choice between physical or logical clocks.

Major problems with physical clocks are left unmentioned in the article, in particular that they may be inconsistent with causality. For example, if a client reads the value for a key in the database, and writes back an updated value, it is possible that the write has no effect. This could happen if the client that previously wrote this key has a fast-running clock, and/or the client making the subsequent update has a slow-running clock, and so the second write's physical timestamp ends up being less than the previous write's timestamp. The article's claim that "clock synchronisation is nice to have in a Cassandra cluster but not critical" is completely wrong: clock sync is in fact crucial.

If a client makes several successive writes to the same key, and the clock jumps backwards due to NTP stepping or a leap second, then it could happen that a later write fails to overwrite an earlier value because the later write has an earlier timestamp. In all of these cases, there is no error (the client believes that the database has stored the data) even though the data has in fact been discarded, not stored. This problem is discussed further in this article: https://aphyr.com/posts/299-the-trouble-with-timestamps

The article explains that vector clocks allow the database to detect concurrently written values, giving the application the option of merging those values without data loss. However, as the article points out, this can be difficult for applications to do correctly. The alternative used by Cassandra, namely last-write-wins, simply discards all but one value when several values are written concurrently. This is certainly simpler, but the potential for data loss is not really acknowledged in this article. Moreover, the LWW conflict resolution strategy can be used even in a system that uses vector clocks, by using a merge function that discards all but one value. Using vector clocks opens the *option* of merging concurrently written values, but it does not *force* the application to be any more complicated than one based on LWW. Moreover, if detection of concurrent writes is not desired, Lamport clocks could be used instead of vector clocks: these would be fine for LWW semantics, and would avoid the problems of physical clocks.

The article claims that Cassandra's approach is faster because an object does not need to be read in order to be updated. However, the choice of logical or physical clocks does not affect this aspect: even with logical clocks, a write that simply overwrites any existing value without reading its value (a so-called *blind write*) can be performed without reading a record first.

The only real reason I can see in favour of physical clocks is that they allow the API for clients to be simpler. If logical clocks are used, a write that depends on a prior read needs to propagate the logical timestamp from the read operation to the write operation, which means that they have to somehow be coupled in the client code. Using physical timestamps does not have this requirement. However, the article doesn't actually make this argument.

In summary, the article does not make a convincing argument in favour of using physical time, and fails to acknowledge its risks. My recommendation for using Cassandra is that it's okay to use in situations where a record is only written once and then never updated (for example, for storing log files, where each line of the log is one record), because in that case the timestamp is irrelevant. However, any applications that update existing records in Cassandra are at risk of data loss if they encounter any problems with their clock synchronisation.

Exercise 18. *Give pseudocode for the ABD algorithm.*

Each replica executes the following:

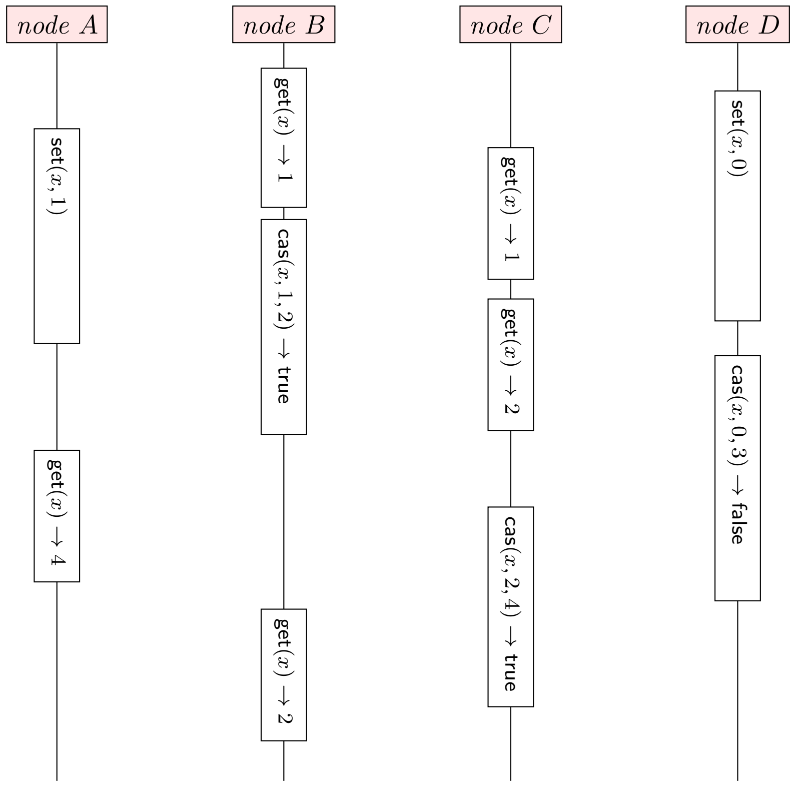
```
on initialisation do
    $values := \{\}$
end on

on receiving $(\mathsf{get}, k)$ from a client do
    send $\{(t, v) \mid (t, k, v) \in values\}$ as response
end on

on receiving $(\mathsf{set}, t, k, v)$ from a client do
```

```
    if $\nexists t', v'.\ (t', k, v') \in \mathit{values} \land t < t'$ then
        $\mathit{values} := \{(t', k', v') \in \mathit{values} \mid k' \neq k\} \cup \{(t, k, v)\}$
    end if
    send ok as response
end on
```

Each client executes the following:

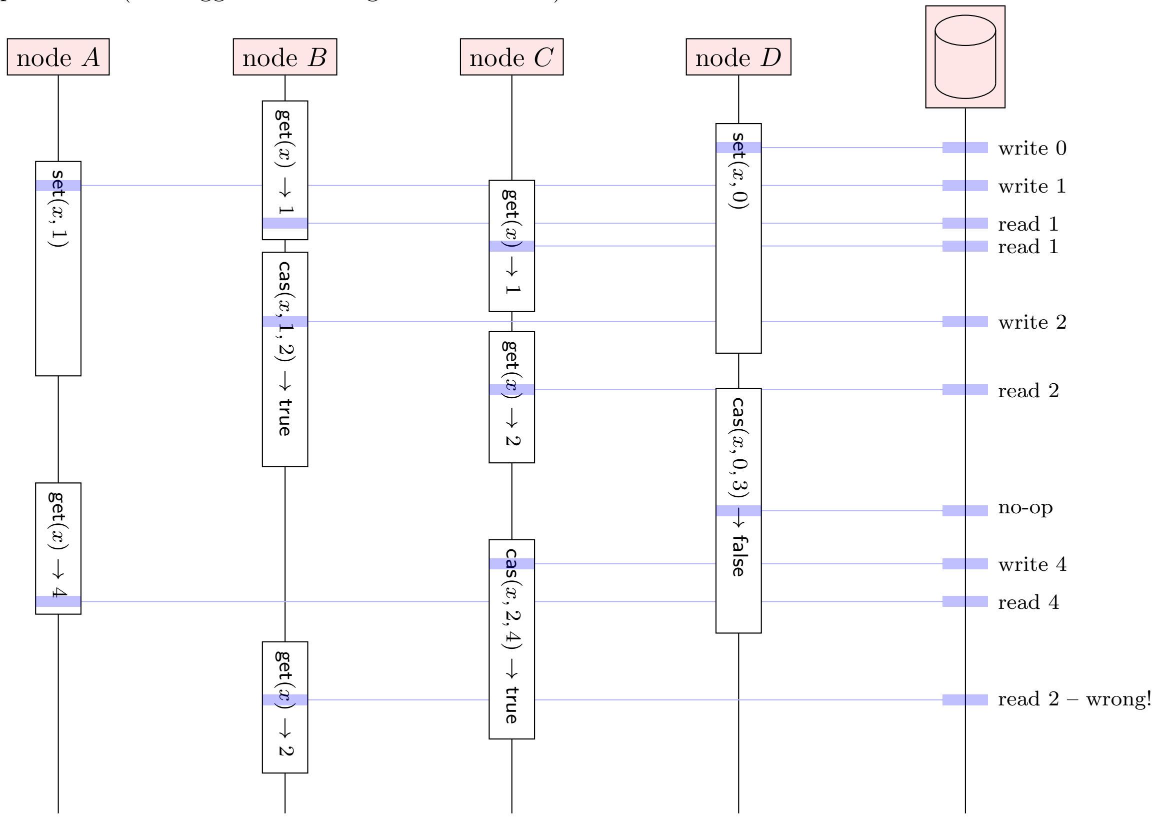
```
on initialisation do
    $\mathit{writes} := 0$
end on

on request to read value for key $k$ do
    send $(\mathsf{get}, k)$ to each replica in $\mathit{replicas}$
    $\mathit{responses} := \{\}$
    while $|\mathit{responses}| < \left\lceil \frac{|\mathit{replicas}|+1}{2} \right\rceil$ do
        receive response $(t, v)$ from replica $i$
        $\mathit{responses} := \mathit{responses} \cup \{(i, t, v)\}$
    end while
    $\mathit{maxt} := \max(\{t \mid \exists i, v.\ (i, t, v) \in \mathit{responses}\})$
    $\mathit{maxv} :=$ the unique value $v$ such that $\exists i.\ (i, \mathit{maxt}, v) \in \mathit{responses}$
    send $(\mathsf{set}, \mathit{maxt}, k, \mathit{maxv})$ to each replica in $\{i \in \mathit{replicas} \mid (i, \mathit{maxt}, \mathit{maxv}) \notin \mathit{responses}\}$
    while $|\{i \mid (i, \mathit{maxt}, \mathit{maxv}) \in \mathit{responses}\}| < \left\lceil \frac{|\mathit{replicas}|+1}{2} \right\rceil$ do
        receive response ok from replica $i$
        $\mathit{responses} := \mathit{responses} \cup \{(i, \mathit{maxt}, \mathit{maxv})\}$
    end while
    return $\mathit{maxv}$
end on

on request to set key $k$ to value $v$ do        ▷ NOTE: only one designated node may perform writes
    $\mathit{writes} := \mathit{writes} + 1$
    send $(\mathsf{set}, \mathit{writes}, k, v)$ to each replica in $\mathit{replicas}$
    $\mathit{responses} := \{\}$
    while $|\mathit{responses}| < \left\lceil \frac{|\mathit{replicas}|+1}{2} \right\rceil$ do
        receive response ok from replica $i$
        $\mathit{responses} := \mathit{responses} \cup \{i\}$
    end while
end on
```

Exercise 19. *Is the following execution linearizable? If not, where does the violation occur?*



It is not linearizable. The violation is the final read by B, which should have returned 4 rather than 2. The value of x was previously 2, but C's CAS$(x,2,4)$ changed it to 4, then A read 4. Because B's final read started after A's final read finished, B's read should have returned a value no older than A's.

The following diagram visualises the points in time at which each operation takes effect, and the resulting total order of atomic operations. Example based on Figure 9-4 from *Designing Data-Intensive Applications* (see suggested reading for this course).



Exercise 20. *Would it be equivalent to combine properties 4 and 5 into a single liveness property, namely*

"If a node broadcasts a message m and does not crash, then every node that does not crash eventually delivers m"?

No, this is not equivalent. Say message m is broadcast by node N_1, then m is delivered by node N_2, and then N_1 crashes. In the original properties given on the slide, m must now be delivered by all non-crashed nodes because it was delivered by one node. In the revised property, other nodes would not necessarily need to deliver m, because it was sent by a crashed node. The revised property therefore does not ensure that all non-crashed nodes deliver the same sequence of messages.

Exercise 21. *Three nodes are executing the Raft algorithm. At one point in time, each node has the log shown below:*

log at node A:

m_1	m_2
1	1

log at node B:

m_1	m_4	m_5	m_6
1	2	2	2

log at node C:

m_1	m_4	m_7	$\leftarrow$ msg
1	2	3	$\leftarrow$ term

(a) Explain what events may have occurred that caused the nodes to be in this state.
(b) What are the possible values of the commitLength *variable at each node?*
(c) Node A starts a leader election in term 4, while the nodes are in the state above. Is it possible for it to obtain a quorum of votes? What if the election was instead started by one of the other nodes?
(d) Assume that node B is elected leader in term 4, while the nodes are in the state above. Give the sequence of messages exchanged between B and C following this election.

a. Node A was leader in term 1; it successfully replicated m_1 to all nodes, then it appended m_2 to its own log, but message loss (or a crash of A) prevented m_2 from being replicated to nodes B and C. Node B suspected A as faulty, was elected leader in term 2 with a vote from C, appended m_4 to its log, and successfully replicated it to C (but not A). Node B then also appended m_5 and m_6 to the log, but the replication requests were lost or B crashed. Finally, node C suspected B to be faulty, and started a leader election in term 3. A has now recovered, and C becomes leader with a vote from A (B would not vote for C in this case, because B's log is more up-to-date than C's). C then appends m_7 to its log and sends it to A and B, but these messages have not arrived yet.

b. At A, *commitLength* must be either 0 or 1, since only the first log entry has been replicated to a quorum of nodes. At B and C, it might be 0, 1, or 2, since the first two entries have been replicated to a quorum. In all cases, the value might be 0 because the acknowledgement messages may not yet have arrived.

c. Nodes B and C would not vote for A because the terms of their last log entries (2, 3) are higher than the term of A's last log entry (1), and so A would not be able to form a quorum. Likewise, C would not vote for B. However, A would vote for B, and both A and B would vote for C. Thus, B or C would be able to form a quorum if it started an election.

d. B sends to C: $(\mathsf{LogRequest}, B, 4, 4, 2, 1, \langle\rangle)$
C sends to B: $(\mathsf{LogResponse}, A, 4, 0, \mathsf{false})$
B sends to C: $(\mathsf{LogRequest}, B, 4, 3, 2, 1, \langle(m_6, 2)\rangle)$
C sends to B: $(\mathsf{LogResponse}, A, 4, 0, \mathsf{false})$
B sends to C: $(\mathsf{LogRequest}, B, 4, 2, 2, 1, \langle(m_5, 2), (m_6, 2)\rangle)$
C sends to B: $(\mathsf{LogResponse}, A, 4, 4, \mathsf{true})$
(The *commitLength* field could be 0, 1, or 2, as per part c. Node C will discard m_7 when it receives m_5 and m_6 from B. This is okay because m_7 was not written to a quorum of nodes, and hence it was not acknowledged as committed.)

Exercise 22. *Prove that the operation-based map CRDT algorithm provides strong eventual consistency.*

The *eventual delivery* property of strong eventual consistency holds because when a message is delivered by one non-faulty node, reliable broadcast ensures that it will also be delivered by all other non-faulty nodes. Therefore, each update applied on one non-faulty replica is also applied by every other non-faulty replica.

To prove the *convergence* property, we must show that any two replicas that have processed the same set of updates are in the same state. Note that the replica state *values* is determined entirely and deterministically by the sequence of messages delivered by reliable broadcast. Consider two replicas, A and B, on which the broadcast protocol has delivered the same set of messages; that is, their sequences of deliveries contain the same messages, and A's sequence of deliveries is a permutation of B's sequence of deliveries. Then we must show that processing the messages in A's delivery order results in the same state as processing the messages in B's delivery order.

We can transform A's sequence of deliveries into B's sequence of deliveries by repeatedly performing pairwise swaps of two adjacent messages in the sequence, since one is a permutation of the other. Let a and b be any two such messages being swapped. To show convergence, we must prove commutativity: that is, processing first a and then b (in any replica state) has the same effect on the state as processing first b then a. If this property holds, then each of the pairwise swaps that we make has no effect on the final state.

To prove commutativity, let $a = (\mathsf{set}, t_a, k_a, v_a)$ and $b = (\mathsf{set}, t_b, k_b, v_b)$. We have $t_a \neq t_b$ due to the assumption of globally unique timestamps and because reliable broadcast does not duplicate messages. Assume $t_a < t_b$ without loss of generality. Let *values* be the state of both replicas immediately before a or b are processed.

Consider first the case where $k_a \neq k_b$. The pseudocode for delivering a set message only reads or modifies tuples in *values* that contain the key in the message. Thus, two messages containing different keys can be processed in either order without affecting the outcome.

In the case where $k_a = k_b$ we consider the following cases:

1. $\nexists t, v.\ (t, k_a, v) \in \mathit{values}$.

   Consider a replica that first processes a, then b. After processing a, the replica state is $\mathit{values} \cup \{(t_a, k_a, v_a)\}$. After processing b, the replica state is $\mathit{values} \cup \{(t_b, k_b, v_b)\}$.

   Next, consider a replica that first processes b, then a. After processing b, the replica state is $\mathit{values} \cup \{(t_b, k_b, v_b)\}$. After processing a, the replica state remains $\mathit{values} \cup \{(t_b, k_b, v_b)\}$ since $t_a < t_b$. Thus, both delivery orders lead to the same state.

2. There exist t, v such that $(t, k_a, v) \in \mathit{values}$. By uniqueness of timestamps, $t \neq t_a$ and $t \neq t_b$. We then consider further cases:

   (a) $t < t_a < t_b$.

   A replica that first processes a will first be in state $\mathit{values} \setminus \{(t, k_a, v)\} \cup \{(t_a, k_a, v_a)\}$, and then in state $\mathit{values} \setminus \{(t, k_a, v)\} \cup \{(t_b, k_b, v_b)\}$ after processing b.

   A replica that first processes b will first be in state $\mathit{values} \setminus \{(t, k_a, v)\} \cup \{(t_b, k_b, v_b)\}$, and then remain in that state after processing a.

   (b) $t_a < t < t_b$.

   A replica that first processes a will remain in state *values* after processing a. After processing b it moves to state $\mathit{values} \setminus \{(t, k_a, v)\} \cup \{(t_b, k_b, v_b)\}$.

   A replica that first processes b will first be in state $\mathit{values} \setminus \{(t, k_a, v)\} \cup \{(t_b, k_b, v_b)\}$, and then remain in that state after processing a.

   (c) $t_a < t_b < t$. Then processing a and b leaves the replica state unchanged, regardless of the order in which they are processed.

In all of these cases, the state of a replica after processing a then b is the same as the state of a replica after processing b then a. Therefore, a and b commute in all replica states, and therefore the algorithm ensures convergence.

Exercise 23. *Give pseudocode for a variant of the operation-based map CRDT algorithm that has multi-value register semantics instead of last-writer-wins semantics; that is, when there are several concurrent updates for the same key, the algorithm should preserve all of those updates rather than preserving only the one with the greatest timestamp.*

Note that this algorithm uses causal broadcast to ensure that when one updates overwrites another, the causally later (overwriting) update is delivered after the update that it overwrites. We extend the broadcast message to include a set T, which is the set of timestamps of causally prior operations that are overwritten by a particular set operation. This way, when a broadcast message is delivered, we know which values were written by operations that happened before this set operation, and which operations were concurrent.

```
on initialisation do
    $values := \{\}$
```

```
end on

on request to read value for key $k$ do
    return $\{v \mid \exists t.\ (t, k, v) \in \mathit{values}\}$
end on

on request to set key $k$ to value $v$ do
    $T := \{t \mid \exists v'.\ (t, k, v') \in \mathit{values}\}$
    $t :=$ newTimestamp()          ▷ globally unique, e.g. Lamport timestamp
    broadcast $(\mathsf{set}, T, t, k, v)$ by causal broadcast (including to self)
end on

on delivering $(\mathsf{set}, T, t, k, v)$ by causal broadcast do
    $\mathit{values} := \{(t', k', v') \in \mathit{values} \mid t' \notin T\} \cup \{(t, k, v)\}$
end on
```