December 2021

Notation

In backtick-quoted code, equality checking, comparisons etc follow the Java style, e.g. `==` for equality, `<=,>=` for comparisons.
In dollar-quoted maths (KaTeX), equality checking, comparisons will follow usual mathematical notation.

Furthermore, a mathematical statement like $x = 1$, would in code form be `x==1`. This is to be taken as a declaration that x equals 1 is a true statement, unless it is explicitly mentioned for e.g., that `x==1` does not hold, or that `x==1` evaluates to false.

Assignment in maths is denoted by $\leftarrow$ or $:=$.

Mutual Exclusion Problem

Properties

- Mutual Exclusion, Progress, No starvation (NOT bounded wait!)
- No starvation implies progress
- Progress and bounded wait implies no starvation

For example, in the following attempt to achieve mutual exclusion between 2 processes, there is bounded wait, but not progress. Additionally, a process can still get starved.

```
// shared variables
let turn: number;

function RequestCS(i: number) {
    assert(i == 0 || i == 1);
    while (turn == (1 - i)) {}
}

function ReleaseCS(i: number) {
    assert(i == 0 || i == 1);
    turn = 1 - i;
}
```

Peterson's Algorithm

Mutual Exclusion Property

In the proof by contradiction, we consider the most recent executions of both processes. For instance, process 0 is in the CS (critical section), so the most recent lines of code executed (barring the CS itself) is

`RequestCS(0)` . Similarly, since process 1 is in the CS, the most recent lines of code executed falls under `RequestCS(1)` .

Note that we did not claim that the most recent executions **across all processes** are `RequestCS(0)` and `RequestCS(1)` . It could, for example be in the following order:

1. P(rocess)0: `RequestCS(0)`
2. P1: `RequestCS(1)`
3. P0: Enter CS
4. P0: `ReleaseCS(0)`
5. P0: `RequestCS(0)`
6. P0: Enter CS
7. P1: Enter CS
   So, clearly, the most recent lines executed outside of the CS are actually `ReleaseCS(0)` and `RequestCS(0)` .

But the point is, when P0 and P1 are in the CS, the most recent lines executed by P_i must be `RequestCS(i)` .

With this detail established, we continue on with the proof.

The variable `turn` can only take the values in $\{0,1\}$, and we can prove this since `turn` is initialized to $0 \in \{0,1\}$ and in the program, `turn` can only ever be assigned to `0` or `1` .

We transcribe the proof given in lecture slides. By symmetry, we only discuss the case where `turn == 0` . (Note that when we say P0 executed `turn=1` we are referring to the most recent execution of that line by P0; similarly for other commands.)

Before reaching the CS, P0 must first execute `turn=1` and P1 must first execute `turn=0` . Since `turn==0` , of these 2 assignments, the (globally) most recent assignment must be `turn=0` by P1. Hence, between the time P1 executes `turn=0` and **now**(i.e. both P0, P1 in CS), P0 will not execute any line before `turn=1` (inclusive).

Hence, when P0 reaches the while loop, we are assured that `turn==0` as there are no further assignments to `turn` . Furthermore, for P1 to enter the CS, we must have `wantCS[0]==true` evaluating to false, i.e. `wantCS[0]==false` .

Here, we introduce arrow notation as follows. Event A $\rightarrow$ event B means that A took place prior to B.
P0 `wantCS[0]=true` $\rightarrow$ P0: `turn==1` $\rightarrow$ P1: `turn=0` $\rightarrow$ P1: while loop
However, based on this timeline analysis, we see at the time of P1's while loop the most recent assignment to `wantCS[0]` was by P0, setting it to `true` , a contradiction. $\square$

Progress Property

Suppose at least one of P0 and P1 are waiting. Then the only reason P0 and P1 may not enter the CS is due to the while loop, since all other lines in `RequestCS` can be run in constant (in particular, bounded) time, i.e. it suffices to consider the case where a process is either

1. at the while loop if waiting,
2. or completely outside `RequestCS` if not waiting.

Consider cases, where we split the cases by how many processes are waiting. The lecture slides seem to only consider the first case.

Case 1: Both processes are waiting.
Progress holds by the fact that it is not possible for `turn==0` and `turn==1` to both be true.

Case 2: Only P0 is waiting for an indefinite period of time. Note our use of "indefinite", so that we can assume that P1 is not between the start of `RequestCS` and the end of `ReleaseCS` for all but a finite amount of time. We claim that if P1 is outside of this `RequestCS --- ReleaseCS` section, then `wantCS[1]==false`, so that P0 can pass the while loop and enter the CS. This is true since `wantCS[1]` is initialized to `false` and whenever P1 enters the `RequestCS --- ReleaseCS` section, P1 always sets `wantCS[1]` back to `false` in `ReleaseCS`, so that when P1 exits, `wantCS[1]` is `false`.

Case 3: Symmetric to Case 2.

Hence progress. $\square$

Remark When we say that a process is trying to enter the CS, a possible way of rigorously defining this would be that the process is within the `RequestCS` function. We can apply this definition to cases 2 and 3 above. For e.g. in case 2, when we assume that only P0 is waiting, we do not allow P1 to be in the `RequestCS(1)` function. P1 is allowed to be in the CS, as well as in `ReleaseCS`, however. This definition would make reasoning easier, as we do not have to deal with "all but finite".

No Starvation Property

WLOG, we show that if P0 is waiting, P0 will not be starved. It suffices to show that $\exists k \in \mathbb{N}$ such that at most k *other* processes distinct from P0 (in this case P1) that can enter the CS before P0.

Note that we do not need to prove the case where P0 is the only process waiting, since progress implies P0 must be able to enter CS. Hence, we now prove bounded wait.

There are some subtleties here. We actually need to establish a reference timepoint with respect to the bound on number of times P1 enters CS. Our reference timepoint here will be the time at which P0 is at the while loop.

Similarly to the start of the proof of progress, we assume that P0 is at the while loop, where the loop condition evaluates to `true`. We claim that $k = 2$ will suffice for the while loop to evaluate to false, allowing P0 to return from `RequestCS`.

When P0 is waiting, P0 will not make any changes to the `turn` variable. While it may seem that $k = 1$ is good enough, to be very rigorous, we should say $k = 2$. This is because the first time P1 enters the CS, P1 may already be at the `while` loop, below the line `turn=0`. (So usually, $k > 1$. We can also define bounded wait to only include number of times other processes can enter CS **starting from RequestCS**. In that case, then $k = 1$ would be fully correct.)

We now consider the 2nd time P1 tries to enter CS, starting from calling `RequestCS` , thereby setting `turn = 0` , in particular, P1 can only set change `turn` to `0` . The next time P1 calls `RequestCS` again, `turn == 0 && RequestCS[0] == true` so that P1 will block at the while loop. Hence, as long as P0 continues to run at this new timeframe, P0 can enter the CS.

Remark: It is actually not important whether P1 blocks or not (since this is relevant to mutual exclusion, but we are only talking about starvation here). As long as `turn == 0` , P0 will pass the while loop. $\square$

Lamport's Bakery Algorithm

We note the following in the algorithm.

- Tuples `(queueNumber, id)` are ordered first by `queueNumber` , then `id` .
- In "get a number", $number[myid] \leftarrow \max\{number[j] : 0 \leq j < n\} + 1$

Progress Property

Suppose a non-zero number of processes, say $1 \leq k \leq n$ are waiting at `RequestCS` . Examining the code, we see that a process can only wait at one of the while loops in the second for loop.

Note that if a process is not waiting, i.e. $P \notin \{P_{i_1}, \ldots, P_{i_k}\}$, we can simply assume that it is completely outside `RequestCS` such that `choosing[process_id]==false` and `number[process_id]==0` .

Suppose a process, $P = P_i \in \{P_{i_1}, \ldots, P_{i_k}\}$ is one of the waiting processes. We consider cases.

1. P is waiting at `while (choosing[j]==true)` for some j. In this case, process P_j, which we will write as Pj can always complete the "get a number" portion, such that in a finite amount of time, `choosing[j]=false` executes. Then, `choosing[j]==true` evaluates to false under process P, or otherwise, we must have the assignment `choosing[j]=true` , which would mean Pj entered the CS, left the CS, and called `RequestCS` . While the 2nd case should not happen based on the code, even if it does, we already have progress.
2. After considering case 1, we can now assume that *all* waiting processes are now waiting at the second while loop, `while (number[j]!=0 && Smaller(number[j], j, number[i], i)` . In this case, we claim that we can find some $P_i \in \{P_{i_1}, \ldots, P_{i_k}\}$ such that $\forall j \in \{1, \ldots, n\}$, `number[j]!=0 && Smaller(number[j], j, number[i], i)` evaluates to `true` , which we will prove below.

Remark: we can assume *"all waiting processes"* since there are finitely many processes, so that it takes a finite amount of time for all processes that will ever wait in this timeframe to be blocked at the 2nd while loop.

Lemma. $\exists i \in \{1, \ldots, k\}, \forall j \in \{1, \ldots, n\}$, `number[j]!=0 && Smaller(number[j], j, number[i], i)` evaluates to true.

If $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$, then `number[j]==0` and so `number[j]!=0` evaluates to `false` .

Otherwise, amongst the elements `(number[index], index)` indexed by $\{i_1, \ldots, i_k\}$, there must exist a minimal element, since there are finitely many elements. If i_l indexes a minimal element, then process $P = P_{i_l}$ will have `Smaller(number[j], j, number[i_l], i_l)` evaluate to `false` for each $j \in \{1, \ldots, n\}$.

Hence such a process P will pass both while loops at each iteration of the for loop and enter the CS. (in particular, we have shown that at least one waiting process can enter the CS.) $\square$

No Starvation Property

We fix a process of id $i \in \{1, \ldots, n\}$, and that $P = P_i$ is waiting to enter the CS. We claim that P will not be starved.

As proven in progress, P will not be blocked by `while (choosing[j] == true)`.

At the current point of discussion, we sort the elements $\{(number[j], j) : 1 \leq j \leq n \wedge number[j] \neq 0\}$. Then suppose i is at position p. Then we claim that at most (in fact, exactly) $p - 1$ processes can enter the CS before P does, i.e. $k = p - 1$. (This is an inductive argument)

WLOG, we assume that all $p - 1$ processes ranked ahead of i by our ordering have entered and left the CS. At this point, regardless of what the other processes do, $(number[i], i) = \min\{(number[j], j) : 1 \leq j \leq n \wedge number[j] \neq 0\}$, since "get a number" will grant any process entering `RequestCS` a queue number $\geq number[i] + 1$.

Hence, P will not be blocked by `while (number[j]!=0 && Smaller(number[j], j, number[i], i)` for any $j \in \{1, \ldots, n\}$. $\square$

Mutual Exlusion Property

Suppose more than one process is in the CS. Choose any two of them, and label them P0, P1. Let the index of P0 be k and the index of P1 be l. WLOG, $(number[k], k) < (number[l], l)$.

Consider the timeframe where P1 is at the second while loop, in particular, when P1 is comparing against P0. Since P1 passes this while loop, either `number[k] == 0` is true or (`number[k] != 0` true, `Smaller(number[k], k, number[l], l)` is false).

1. `number[k] != 0` true and `Smaller(number[k], k, number[l], l)` is false: This case is not possible since $(number[k], k) < (number[l], l)$, in the event that P0 is at its most recent call of `RequestCS`, plus P0 has executed `if (number[k] > number[l])`. Then P1 must get stuck here.
   If P0 has not reached this point, as long as P1's `number[l]` remains the same, the new number that P0 gets must be strictly greater than `number[l]`, because P0's number will be $\geq number[l] + 1$, a contradiction.
2. `number[k] == 0`: At this timeframe, P0 must be before the line `number[myId]++`, since this line will result in `number[k] > 0`. Furthermore, P0 must be in the loop, with loop variable `j > l` (or more precisely, P0 has executed the check `if (number[j] > number[myId])` where `j==l`). If not, P0 will get a number strictly greater than `number[l]`.

Next, consider an earlier timeframe, where P1 is at the first while loop. We further note that P1 manages to pass the first while loop, so that `choosing[k] == false`. There is only 1 possibility for this to occur, this is because `choosing[k]` can only evaluate to true within the "get a number" section. P0 must be at some point strictly (chronologically) before the most recent invoation of the `choosing[myId] = true` line, this also takes into account earlier invocations of `RequestCS(0)` (e.g. where P0 is after `choosing[myId] = false`). We conclude that P0 must then evaluate `number[j] > number[myId]` with $j = l, myId = k$ and `number[k]`. Contradiction. $\square$

Synchronization Primitives

Useful link: https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again

From javadocs on Object

```
public final void notifyAll()
```

Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.
The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. *The awakened threads will compete in the usual manner with **any other threads** that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.*

- In other words, when notify or notifyAll are called, we can view this as the awakened thread(s) moving from the waiting queue to the monitor queue.

Monitor (Java)

A process/thread can be in one of 3 states with regard to a synchronized method (monitor)

1. The thread is actively running. The monitor ensures that at most 1 thread can be in this state.
2. The thread is blocked. This can happen in 2 places.
    1. The thread calls the synchronized method and is attempting to acquire lock.
    2. The thread has been waken up from waiting state and is attempting to reacquire lock.
3. The thread is waiting, i.e. after a call to `wait`, the thread joins the waiting queue.

Useful link: https://stackoverflow.com/questions/5798637/is-it-safe-to-call-a-synchronized-method-from-another-synchronized-method

```
void synchronized method1() {
    method2()
}

void synchronized method2() {
}
```

equivalent to

```
void method1() {
    synchronized (this) {
        method2()
    }
}

void method2() {
    synchronized (this) {

    }
}
```

Assume both methods are in the same class, so that `this` refers to the same object instance.
In a thread, when `method1` calls `method2`, the lock associated with `this` is already acquired, so the thread will not block.

Producer/Consumer Problem

Pseudo-code from lecture slides

```
Object sharedBuffer;

void produce() {
    synchronized (sharedBuffer) {
        if (sharedBuffer.isFull())
            sharedBuffer.wait();
        boolean wasEmpty = sharedBuffer.isEmpty();
        addItemTo(sharedBuffer);
        if (wasEmpty)
            sharedBuffer.notify();
    }
}

void consume() {
    synchronized (sharedBuffer) {
        if (sharedBuffer.isEmpty())
            sharedBuffer.wait();
        boolean wasFull = sharedBuffer.isFull();
        removeItemFrom(sharedBuffer);
        if (wasFull)
            sharedBuffer.notify();
    }
}
```

We analyze this by cases.

One consumer, one producer

We claim that the solution is correct. We need to show that

1. When the buffer is full, producer does not write to it.
2. When the buffer is empty, consumer does not read from it.
3. Progress: Consumer can read from a non-empty buffer. Producer can write into a non-full buffer.

Suppose producer added an item to a full buffer, i.e. the producer executed

```
        addItemTo(sharedBuffer);
```

when the buffer is full. (Call this time T1) We consider an earlier time point T2 where the producer most recently evaluated `sharedBuffer.isEmpty()` . Consider cases.

1. Suppose the evaluation is `false` , then somehow, an item got added into the buffer (between T2 and T1) without the producer having done anything. This is impossible, since only the producer is capable of adding items, and we only have one producer.
2. Suppose the evaluation is `true` , then the producer would enter the wait queue. The only way for the producer to wake up from the wait queue is if the consumer process calls `sharedBuffer.notify();` . We shall study whether it is possible for the buffer to still be full at this point in time.
   Denote T3 as the time the consumer calls `sharedBuffer.notify();` .

Denote T4 as the time the consumer last calls `removeItemFrom(sharedBuffer);` . Note that T4 < T3 < T1. T2 is somewhere before T3.
We claim that T2 < T4. Suppose not, then T4 < T2, but this says that $T4 < T2 < T3 < T1$, which implies that there are 2 processes inside the monitor-locked region, since consumer process is not blocked (since `removeItemFrom(sharedBuffer);` is strictly after the `wait`). Contradiction.
Hence, $T2 < T4 < T3 < T1$. In other words, an item was removed from the buffer at T4, and the buffer remains untouched since then. In particular, the buffer is non-full.

We have a contradiction in both cases. Hence, it is impossible for a producer to produce into a full buffer.

Proving that the consumer does not consume from an empty buffer should be similar.

Progress
Next, we prove progress for producers. Claim: When the buffer is non-full, the producer can write to it in finite time, regardless of what the consumer does.
Clearly, if the producer is not blocked in the wait queue, then the producer can definitely produce for a non-full buffer. Hence, in order to obtain a contradiction, we assume that the producer is somehow in the wait queue despite a non-full queue.

Define T0 as the time point where the producer is in the wait queue, and *the buffer is not full*.
Define T1 as the time point where producer last executed `if (sharedBuffer.isFull())` . This must have evaluated to `true` , since the producer is now blocked.
So between T1 and T0, something must have happened to the buffer such that it is not full, i.e. a consumer removed an item from the buffer at some time T2, where $T1 < T2 < T0$.
Whether the consumer process then notifies the producer on the wait queue depends on whether `if (wasFull)` evaluates to `true` . Hence, define T3 as the time point where consumer executes `boolean wasFull = sharedBuffer.isFull();` . If $T1 < T3$ we are done.

Suppose that $T3 < T1$, this is not possible since we have 2 active processes inside a monitor.

```
Object sharedBuffer;

void produce() {
    synchronized (sharedBuffer) {
        if (sharedBuffer.isFull()) // T1
            sharedBuffer.wait(); // T0
        boolean wasEmpty = sharedBuffer.isEmpty();
        addItemTo(sharedBuffer);
        if (wasEmpty)
            sharedBuffer.notify();
    }
}

void consume() {
    synchronized (sharedBuffer) {
        if (sharedBuffer.isEmpty())
            sharedBuffer.wait();
        boolean wasFull = sharedBuffer.isFull(); // T3
        removeItemFrom(sharedBuffer); // T2
        if (wasFull)
            sharedBuffer.notify();
    }
}
```

To be precise,

- producer is active in the time range $[T1, T0]$ and
- consumer is active at the time range $[T3, T2]$
- where $T1 \in [T1, T0] \cap [T3, T2]$

  there is an non-empty intersection in their active time periods, which contradicts the mutual exclusion guarantee of a monitor.

One consumer, multiple producers

A common point of failure with regard to the usage of monitors the that `notify` has no control over which process is woken up. Furthermore, the process that is woken up also competes with incoming processes.

We come up with a situation in which the synchronization mechanism fails.

Suppose the buffer size is $n \in \mathbb{Z}^+$. Assume that the buffer is full, after producers have written to it.

- 1 producer P1 gets blocked in the wait queue
- 1 consumer enters, removes item, and notifies.
- producer P1 wakes up, but does not manage to acquire monitor lock. Instead, an incoming producer P2 enters and acquires lock. P1 now waits in the monitor queue.
- P2 sees that `sharedBuffer.isFull() == true` and produces for the buffer. *Note that the buffer is now full again.*

- P2 exits monitor, P1 who is in the monitor queue then dequeues and adds another item to the queue. The algo therefore fails.

One particular point of failure is the use of `if` statement instead of `while` .

A similar case can be made for the multiple consumer, one producer case.

Change: Change the `if` to `while`

```
Object sharedBuffer;

void produce() {
    synchronized (sharedBuffer) {
        while (sharedBuffer.isFull())
            sharedBuffer.wait();
        boolean wasEmpty = sharedBuffer.isEmpty();
        addItemTo(sharedBuffer);
        if (wasEmpty)
            sharedBuffer.notify();
    }
}

void consume() {
    synchronized (sharedBuffer) {
        while (sharedBuffer.isEmpty())
            sharedBuffer.wait();
        boolean wasFull = sharedBuffer.isFull();
        removeItemFrom(sharedBuffer);
        if (wasFull)
            sharedBuffer.notify();
    }
}
```

We now attempt a different way to make the algo fail. Consider the following sequence of events. Note that the buffer is of capacity n.

- There are $n + 2$ producers. n producers write to the buffer, the remaining 2 producers waits in the queue.
- There are 2 consumers. The first consumer wakes up a producer P1. But when P1 wakes up, it does not manage to acquire lock. Instead, the second consumer acquires lock first.
- Now, 1 producer P2 is in the monitor queue, despite the fact that the buffer is not full (since P1 produces 1 item, and the 2 consumers consume 2 items)

The point of failure is that `notify` only wakes 1 process up. (i.e. one of the 2 sleeping producers). Only the first consumer manages to call notify. The second one does not since buffer is no longer full.

Change: Change `notify` to `notifyAll`

```
Object sharedBuffer;

void produce() {
    synchronized (sharedBuffer) {
        while (sharedBuffer.isFull())
            sharedBuffer.wait();
        boolean wasEmpty = sharedBuffer.isEmpty();
        addItemTo(sharedBuffer);
        if (wasEmpty)
            sharedBuffer.notifyAll();
    }
}

void consume() {
    synchronized (sharedBuffer) {
        while (sharedBuffer.isEmpty())
            sharedBuffer.wait();
        boolean wasFull = sharedBuffer.isFull();
        removeItemFrom(sharedBuffer);
        if (wasFull)
            sharedBuffer.notifyAll();
    }
}
```

This is probably correct. Idk.

Change: Remove the conditional

```
Object sharedBuffer;

void produce() {
    synchronized (sharedBuffer) {
        while (sharedBuffer.isFull())
            sharedBuffer.wait();
        addItemTo(sharedBuffer);
        sharedBuffer.notify();
    }
}

void consume() {
    synchronized (sharedBuffer) {
        while (sharedBuffer.isEmpty())
            sharedBuffer.wait();
        removeItemFrom(sharedBuffer);
        sharedBuffer.notify();
    }
}
```

While this may seem correct at first, these two are not equivalent.
i.e. `if (wasEmpty) { sharedBuffer.notifyAll(); }` and `sharedBuffer.notify();` . The main problem, again, is that notify has no control over which process is woken up, so the producer calling `notify` is not guaranteed to

wake up a consumer process. The producer may wake up another waiting producer, and we can easily construct a concrete sequence of events to demonstrate failure.

Assume that the shared buffer of size n is full.

- $2n$ producers enter and block.
- $2n$ consumers enter. n of them wake up n producers. The other n block.
- Now there are n producers and n consumers in the queue, and another n producers that are awake.
- The n awake producers produce items for the buffer and make it full again. However, unfortunately, when `notify` is called, only producers are woken up.
- We now have a full buffer, n awakened producers and n waiting consumers.
- The awakened producers would go back to the waiting queue as `sharedBuffer.isFull()` evaluates to `true` .
- We now have a full buffer, n waiting producers and n waiting consumers. Despite the full buffer, all the consumers are waiting.

Hence, this fails.

Reader-Writer problem

To prove correctness, we need to show the following:

1. When a writer is writing to a file F, no reader can read, and no other writer can write.
2. When (possibly multiple) readers are reading, writers cannot write.

More lemmas to prove:

- When reader calls `notify` , there can be no reader in the wait queue.

Extensions:

- Devise a no starvation algorithm.

```
void writeFile() {
    synchronized (object) {
        while (numReader > 0 || numWriter > 0)
            object.wait();
        numWriter = 1;
    }
    // write to file;
    synchronized (object) {
        numWriter = 0;
        object.notifyAll();
    }
}

void readFile() {
 synchronized (object) {
        while (numWriter > 0)
            object.wait();
        numReader++;
    }
    // read from file;
    synchronized (object) {
        numReader--;
        object.notify();
    }
}
```

Correctness

We prove **1**. Suppose a writer W1 is writing to file F. Consider cases.

1. Suppose another writer W2 is writing to F as well. We define time point T1 where W1 most recently executes `while (numReader > 0 || numWriter > 0)`, and time point T2 such that W2 most recently executes `while (numReader > 0 || numWriter > 0)`. WLOG, $T1 < T2$. Then right before T2, `numWriter == 1`, since T2 can only occur when W1 has left the monitor block, so that `numWriter = 1` has executed. Hence, W2 will be blocked by the while loop and join the wait queue instead. Contradiction.
2. Suppose another reader R is reading from F. We define time point T1 where W1 most recently executes `while (numReader > 0 || numWriter > 0)` and time point T2 where R most recently executes `while (numWriter > 0)`. We consider cases.
    1. If $T1 < T2$, this is not possible since `numWriter == 1` at the time of T2 and reader R will get blocked.
    2. If $T2 < T1$, this is also not possible since `numReader++` has been executed by R by the time of T1. Hence, W1 gets blocked at T1. There is a small detail here. We need to show that `numReader` is always non-negative, so that when `numReader++` executes, it becomes positive. To show this, we observe that we can pair up the `++` and `--`.
    We have thus attained a contradiction in both subcases.
    Hence, we have proven **1**. $\square$

We prove **2**. Suppose at least 1 reader R is reading, and a writer W is writing to F. This is actually the same as case 2 of **1**. So we have this proven for free.

Lemma When reader calls `notify`, there can be no reader in the wait queue.
Let R1 be the reader calling `notify` at time point T1 and we suppose that at T1 there exists a reader R2 in the wait queue.

```
void writeFile() {
    synchronized (object) {
        while (numReader > 0 || numWriter > 0)
            object.wait();
        numWriter = 1;
    }
    // write to file;
    synchronized (object) {
        numWriter = 0;
        object.notifyAll();
    }
}

void readFile() {
 synchronized (object) {
        while (numWriter > 0) // T2: R2, T3: R1
            object.wait(); // R2 waiting
        numReader++;
    }
    // read from file;
    synchronized (object) {
        numReader--;
        object.notify(); // T1
    }
}
```

Let T2 be the time R2 most recently executed `while (numWriter > 0)` and let T3 be the time R1 most recently executed `while (numWriter > 0)`. We consider cases. Note that it is clear that $T2, T3 < T1$.

1. Suppose $T3 < T2 < T1$. At T3, `numWriter == 0` since R1 passes the while loop and at T2, `numWriter > 0` since R2 get blocked. Hence, this says that there is an incoming writer W in the time range $(T3, T2)$. Let T4 be the time W entered the monitor. We see that $T3 < T4 < T2$. W, seeing that `numReader > 0`, enters the wait queue and is unable to increment `numWriter`. The same holds even if more than 1 writer enters. Hence, it is impossible that `numWriter > 0`. Contradiction.
2. Suppose $T2 < T3 < T1$. At T2, `numWriter > 0` and at T3, `numWriter == 0`. Let W be a writer present in the time interval $(T2, T3)$. W writes and executes `notifyAll`. It is possible that another writer wakes up and acquires lock, but WLOG, we assume that W is the last writer to acquire lock **before T3**, i.e. after this it must be a reader that acquires lock. Let T4 be the time point W exits the monitor. At this point, no process can be in the wait queue, since all are woken up by `notifyAll`. In particular, R2 is awake. Since the next processes that acquire lock are all readers, this also includes R1. When R1 eventually reaches

T1, there are no readers in the wait queue. This completes the proof.
$\square$

Corollary If a reader notifies a process, that process must be a writer.

Anti-Starvation Done in homework document.

Consistency Conditions

Various notions of consistency

Based on my understanding, when we say "a particular execution is consistent", we require the execution to conform to a number of things

- The specification of the abstract data type/objects that processes operate on during the execution
  - For e.g., an integer data type may be specified to have
    - Read()/IncrementInRegister()/WriteFromRegister() operations, or
    - Read()/Increment() operations
  - When using a Read()/IncrementInRegister()/WriteFromRegister() integer data type, incrementing may not be atomic. Hence, executing an increment on a Read()/IncrementInRegister()/WriteFromRegister() integer data type may fail the consistency definition of a Read()/Increment() integer data type due to interleaving. (For e.g. incrementing in the register and writing being separate steps)
- With the abstract data type operations specified, the consistency model also needs to be specified.
  - For e.g., we may require the execution to obey sequential consistency.

In particular, we can see that consistency model is only meaningful when we have defined the abstract data types (and the operations that they support, and the semantics of those operations) that the processes work with.

Execution history

We use the following notation, if not otherwise specified
An event e takes the form

$$\texttt{inv/resp(process, operation\_type, resource, value)}$$

- Value may be omitted if no value is involved in the operation

An operation o is a pair of events e, e', where e denotes the invocation event and e' denotes the response event. The properties associated with an operation are retrievable as follows:

- Invocation event $e = inv(o)$
- Response event $e' = resp(o)$

- Process $proc(o)$
- Resource $res(o)$

A history H implies a partial order $<_H$, where $o_1 < o_2 \iff resp(o_1) < inv(o_2)$. (Inequalities between events refer to the physical time at which they occur.)
This is known as the external order, or occurred before order.

Remark Consistency conditions relate to parallel systems, not distributed systems. The difference is that a parallel system is located in a single computer, where it is possible for a single physical clock to dictate what is the physical time. Whereas in a distributed system, there are many computers, some remotely located, so that there are multiple physical clocks with varying degrees of accuracy, so it is near impossible to have a single physical time. In the next chapter, we see that logical clocks are used instead.

Sequential history

Intuitively, a (legal) sequential history is one in which it seems like a single process is creating invoke events and receiving response events.

Proposition These 2 definitions of sequential history $(H, <_H)$ are equivalent.

1. $<_H$ is a total order on operations, such that for all operations o_1, o_2, $o_1 <_H o_2 \vee o_2 <_H o_1$.
2. Every invocation of an operation o, $inv(o)$, must be immediately followed by its response, $resp(o)$, i.e. H is of the form $inv(o_1), resp(o_1), inv(o_2), resp(o_2), \ldots$

Clearly, **2** implies **1**. We shall only prove **1** implies **2**.
Suppose **1**. We then conduct induction on H, let H have n operations, so that the length of H is $2n$. WLOG, let the first entry of H be $inv(o_1)$. (Clearly, the first entry of H cannot be a response event).
Suppose the second entry is not $resp(o_1)$, then it is another invocation event, say $inv(o_2)$. But this means that o_1 and o_2 are not comparable, since it is neither the case that $inv(o_2)$ follows $resp(o_1)$ nor is it the case that $inv(o_1)$ follows $resp(o_2)$. $\square$

Process order

A clearer definition of process order is the sub-partial order imposed by $\bigcup_{\text{process } p} <_{H|p}$.

A legal sequential history S for a sequentially consistent history H preserves process order, but not necessarily external order.

Definitions of Linearizability

Claim: The 2 definitions of linearizability are equivalent.
Reformulation of definition 1: For a history $H = \{o_1, o_2, \ldots, o_n\}$, where o_i denotes an operation, $\exists$ choice function $c : \{1, 2, \ldots, n\}$ such that we have the corresponding "degenerate" history $\{c(o_i) \in [inv(o_i), resp(o_i)] : i \in \{1, 2, \ldots, n\}\}$, where for each i, $inv^*(o_i) = resp^*(o_i) = c(o_i)$, giving a

degenerate legal sequential history $H^* =$
$(inv^*(o_1), resp^*(o_1), inv^*(o_2), resp^*(o_2), \ldots, inv^*(o_n), resp^*(o_n))$.

We call H^* degenerate since it is technically not possible for events to take place at the same time.

Definition 2: A history H for which there exists a legal sequential history S that preserves the partial(external) order $<_H$ given by H. i.e. $<_H \subseteq <_S$.

Note the 3 parts of this definition: S needs to be

1. legal
2. sequential
3. preserves $<_H$

$(\implies)$ Suppose definition 1 is true for a history H. We then construct a sequential history $S_c = ([c(o_i) - \epsilon, c(o_i) + \epsilon])_{1 \leq i \leq n}$ where each operation takes arbitrarily short time $\epsilon > 0$ **AND** $inv(o_i) \leq c(o_i) - \epsilon, c(o_i) + \epsilon \leq resp(o_i)$. Since there are finitely many intervals (n of them), it is possible to choose small enough ϵ so that all intervals do not overlap. Since intervals are disjoint, S_c is sequential.
Furthermore, each interval $[c(o_i) - \epsilon, c(o_i) + \epsilon] \subseteq [inv(o_i), resp(o_i)]$ in terms of time, so that the partial order $<_H$ is preserved. For example, if $a <_H b$ (equivalent to $resp(a) < inv(b)$), then $c(a) + \epsilon < c(b) - \epsilon$ so that $a <_S b$. Hence, the partial order is preserved by S_c.
Like the "degenerate" legal history(call it D), S_c preserves the chronological order of events. In other words, for any operation o, the events that have occurred prior to o in S_c are exactly the same as in D, and in the same order too. The applies for what occurs after. Hence, the legality of D implies the legality of S_c, as relative chronology is the same in D and S_c.
Hence, we have show that definition 2 holds as well.

Remark: We cannot just say D is a sequential history as technically speaking, no 2 events can occur at *exactly* the same time (we can treat this as an axiom), so there is no such thing as an operation with invocation time equalling response time.

$(\impliedby)$ Suppose definition 2 is true for a history H with n operations. Let S be the legal sequential history with the properties mentioned (legal, sequential, preserves $<_H$). Without loss of generality, (by possibly renumbering the operations), S gives the sequence $(o_1, o_2, \ldots, o_n)$, where each o_i subsumes a pair of inv and $resp$ events. We then construct our choice function as follows. In fact, we choose points for operations in ascending order from 1 to n.

At each i, let $d_i := \min\{resp(o_j) - \max\{c(o_{i-1}), inv(o_i)\} : j \geq i\}$, where we define $c(o_0) := -\infty$. Certainly $d_i > 0$. The reason:

- $c(o_{i-1}) < resp(o_j)$ by inductive hypothesis
- $inv(o_i) < resp(o_j)$ since $o_i <_S o_j$ for $j > i$. Note that if there exists $j > i$ such that $resp(o_j) < inv(o_i)$, this would imply $o_j <_H o_i$, yet $o_i <_S o_j$ as we have assumed. This contradicts the fact that $<_H \subseteq <_S$.

Then let $c(o_i) := \max\{c(o_{i-1}), inv(o_i)\} + \frac{d_i}{n}$. Intuitively, this gives us freedom to define the points $c(o_{i+1}) = c(o_i) + \frac{d_i}{n}$, $c(o_{i+2}) = c(o_{i+1}) + \frac{d_i}{n}$ if need be.
Formally, this allows us to re-establish the inductive hypothesis, as

$$c(o_i) = \max\{c(o_{i-1}), inv(o_i)\} + \frac{d_i}{n} < \max\{c(o_{i-1}), inv(o_i)\} + d_i \leq \min\{resp(o_j) : j > i\}$$

We then need to show this is legal. But the legality of this comes directly from the legality of S, since the order of operations of the degenerate history S_c exactly follows that of S, precisely stated as $<_{S_c} = <_S$. So we are done. $\square$

Example (Slide 22):
Let x be an integer data type supporting read and write.

- inv(Q, read, x) q
- inv(P, write, x, 1) p
- resp(Q, read, 1)
- inv(R, read, x) r
- resp(P, write)
- resp(R, read, 0)

By rearranging the operations $r <_S p <_S q$ sequentially, we see that this history is sequentially consistent. However, the history is not linearizable. Suppose a linearization S' exists. Then we must have

- $q <_{S'} r$ since $q <_H r$
- $p <_{S'} q$ since p is the only write operation that writes 1

Hence, we get $p <_{S'} q <_{S'} r$. But this is not legal based on the sequential semantics of x, as we are reading 0 after 1 is written.

Sequential consistency is not a local property

Given the events of history H as follows:

1. inv(P, enqueue, x, 1) p_1
2. resp(P, enqueue, x)
3. inv(Q, enqueue, y, 2) q_1
4. resp(Q, enqueue, y)
5. inv(P, enqueue, y, 1) p_2
6. resp(P, enqueue, y)
7. inv(Q, enqueue, x, 2) q_2
8. resp(Q, enqueue, x)
9. inv(P, dequeue, x) p_3
10. inv(Q, dequeue, y) q_3

11. resp(P, dequeue, x, 2)
12. resp(Q, dequeue, y, 1)
    where p_i, q_i denote operations.

As seen in lecture, $H|x$ and $H|y$ are sequentially consistent. Now, we prove formally that H is not sequentially consistent. Suppose not, such that there exists a legal sequential history S for H.

Since S must preserve process order, we must have $p_1 < p_2 < p_3 \wedge q_1 < q_2 < q_3$ -- (0). (Here $<$ denotes $<_S$)

- p_1 enqueues 1 into x, and q_2 enqueues 2 into x. Since p_3 dequeues 2 from x, we must have $q_2 < p_1$ -- (1)
- By a similar argument for y, we must have $p_2 < q_1$ -- (2)

We now have the following ordering which respects (0) and (1)

$$q_1 < q_2 < p_1 < p_2 < p_3$$

But immediately we see a contradiction, since here, $q_1 < p_2$, yet (2) states otherwise. This contradicts the fact that S is a total order.

Hence S cannot exist, that is, H is not sequentially consistent.

Linearizability is a local property

i.e. A history H is linearizable **iff** for all objects x, $H|x$ is linearizable.

Using Definition 1

$(\implies)$ Given history H, suppose that it is locally linearizable. Let $x_1, x_2, \ldots, x_m$ be the objects that appear in H. Restating the assumption, $\forall i \in \{1, \ldots, m\}, H|x_i$ is linearizable, such that there is a choice function $c_i = o \mapsto c_i(o) \in [inv(o, x_i), resp(o, x_i)]$.

Let c be the "union" of all c_i. Now it is not necessarily the case that c is injective for all operations o in H. But that can be easily resolved. For any 2 o_1, o_2 such that $c(o_1) = c(o_2)$, we "perturb" them arbitrarily, e.g. $c(o_1) := c(o_1) - \epsilon, c(o_2) := c(o_2) + \epsilon$ or vice versa, for some arbitrarily small ϵ. This modified c ensures that there are no clashes, i.e. c is injective.

We claim that c is now a legal (degenerate) linearization of H.

We note that the state of an object x during an operation o is solely dependent on:

1. the operations conducted on x prior to o, and
2. the parameters passed to $inv(o)$.

Both of these don't change when we merge all the c_i together to form c. For example, suppose $c_1(o_1) < c_1(o_2)$ and $c_2(o_3) < c_2(o_4)$ are 2 local linearizations of $H|x_1, H|x_2$ respectively. Suppose that merging them gives $c(o_1) < c(o_3) < c(o_2) < c(o_4)$, i.e $c(o_3)$ cuts between $c(o_1)$ and $c(o_2)$. But since all operations are unchanged, in particular, o_1, o_3 still pass the same parameters to x_1, the behavior of x does not change just because there is o_2. - $(*)$
See lecture slide 29 for a similar explanation.

Hence, c is a legal choice function, and by definition 1, H is linearizable.

$(\Longleftarrow)$ Conversely, suppose that H is linearizable. Let c be the choice function. For each object x, let c_x be the restricton of c to those operations involving object x. Similar to $(*)$, we see that restricting c removes those operations that don't involve x, and because they don't involve x, their removal does not affect the legality of the sequence of events involving x. This proves legality, and hence H is locally linearizable. $\square$

Using Definition 2

The lecture has proven the forward direction, so we will prove the converse.

$(\Longleftarrow)$ Suppose H is linearizable, with the linearization being S. We claim that for each object x, $S|x$ is a linearization of $H|x$.

First, since S is a sequential history, so is $S|x$.
Next, $<_H \subseteq <_S$, so $<_{H|x} \subseteq <_{S|x}$. To see this, suppose $o_1 <_{H|x} o_2$, then $o_1 <_H o_2$, then $o_1 <_S o_2$, and finally $o_1 <_{S|x} o_2$.
Finally, S is legal, so $S|x$ is also legal. The idea is again very similar to $(*)$ in the previous section on Using Definition 1. $\square$

Here, we discuss some aspects of the lecture's proof in the forward direction.
Lemma Given a directed acyclic graph G and a topological sort of graph T. Let $<_G$ be the partial ordering of vertices given by G. $<_T$ is then a total order such that $<_G \subseteq <_T$.

The proof is immediate from the definition of a topological sort. $\square$

We discuss how the topological sorting of the operation graph gives a linearization of H. Let S be the sequential history given by the topological sorting.

First, S is equivalent to H since S contains the exact same set of vertices.
Second, S is legal as S merely "splices" together several per-object sequential histories $S|x$, and as the previous section on Using Definition 1 has shown, if $\forall x, S|x$ legal, S is legal.
Third, $<_H \subseteq <_G$ where $<_G$ is the partial order formed by the union of $<_H \cup \bigcup_{\text{object } x} <_{S|x}$ (G here refers to the directed acyclic graph) and $<_G \subseteq <_S$ since S is formed by topologically sorting G. Hence $<_H \subseteq <_S$ and S respects the operational partial order.

Hence S is a linearization of H. $\square$

Next, we give a detailed proof on why the graph is acyclic. Suppose not, such that there exists a cycle in the graph. We start by considering what sort of edges this cycle consists of.

We denote $S|x$ as the linearization of $H|x$. Note that as of this point, we don't really care what is S, in fact, we haven't even defined S. $S|x$ is purely a notation.

1. The cycle does not contain any edges from any $S|x$. i.e. all edges in the cycle are across different objects. But this means that the partial order $<_H$ has a cycle, which is a contradiction.
2. The cycle does not contain any edges across operations on different objects. This implies that the cycle only has edges in a single $S|x$. But $S|x$ is a sequential history, in particular $S|x$ is a history (i.e. $<_{S|x}$ is a partial order), and cannot have cycles.
3. The cycle contains edges from both $S|x$ for various x and across operations on different objects.

Since we have eliminated cases 1 and 2, only case 3 remains. As mentioned in the lecture, a cycle must be of the form

- edges in $S|x_1$
- edges across operations on different objects
- edges in $S|x_2$
- edges across operations on different objects
- ...
- edges in $S|x_n$
- edges across operations on different objects, closing the cycle

where $n \in \mathbb{Z}^+$.

Lemma A very trivial one. Suppose $<_H \subseteq <_S$ where $<_H$ is a partial order and $<_S$ is a total order. Then $<_S$ cannot "disagree" with $<_H$.

Suppose not, such that there exist a, b, $a <_H b$ but $b <_S a$. Then $a <_S b \wedge b <_S a$, so that $a = b$. This is a contradiction. $\square$

The above lemma justifies why multiple edges in $S|x_i$ for any i in the cycle can be contracted to a single one.

For example, suppose for some x, in $S|x$, edges $o_1 \to o_2 \to o_3$ belong to the cycle. We contract this to just $o_1 \to o_3$, equivalently, $o_1 <_{S|x} o_3$. Since $<_{S|x}$ cannot disagree with $<_{H|x}$, we cannot have $o_3 <_H o_1$. Equivalently, in terms of timestamps, we cannot have $resp(o_3) < inv(o_1)$, and this is precisely used in the timeframe analysis later on.

We transcribe the timeframe analysis in lecture precisely. This only considers a specific case where $n = 2$. Suppose we have conducted the aforementioned contraction, and obtained the following cycle:
$D <_H A <_{S|x_1} B <_H C <_{S|x_2} D$

- Since $D <_H A$, $resp(D) < inv(A)$

- Since $A <_{S|x_1} B$, $\sim (resp(B) < inv(A))$ where $\sim$ denotes logical negation. This implies $inv(A) \leq resp(B)$, which implies $inv(A) < resp(B)$.
- Since $B <_H C$, $resp(B) < inv(C)$.
- Since $C <_{S|x_2} D$, $inv(C) < resp(D)$.

Combining these time orderings, we get

$$resp(D) < inv(A) < resp(B) < inv(C) < resp(D)$$

which is a contradiction as by transitivity $resp(D) < resp(D)$.

We summarize one important lemma used above concisely.
Lemma Let H be a history, and let S be a sequential history that respects H so that the partial order $<_H$ is a subset of the total order $<_S$.
Then,

- $o_1 <_H o_2 \implies resp(o_1) < inv(o_2)$
- $o_1 <_S o_2 \implies o_2 \not<_H o_1 \implies resp(o_2) \not< inv(o_1) \implies inv(o_1) < resp(o_2)$

Proposition Linearization may not be unique.
We can consider an example where there are 2 processes A, B, with operations a, b respectively that are not comparable, e.g.
$inv(a), inv(b), resp(a), resp(b)$

Furthermore, the operations a, b act on distinct objects, so both of these linearizations are legal.

- $inv(a), resp(a), inv(b), resp(b)$
- $inv(b), resp(b), inv(a), resp(a)$

Equivalence of definitions of linearizability (A second proof)

We transcribe the suggested solution. Showing definition 1 $\implies$ definition 2 is straightforward, so we only show definition 2 $\implies$ definition 1.

Given a linearization (equivalent sequential history preserving external order) S of H, we construct the following set of linearization points.
Let

$$T = \min\{|inv(o_i) - resp(o_j)| : o_i, o_j \in S\} > 0$$

WLOG, by possibly renumbering the operations, we can consider S as $inv(o_1), resp(o_1), inv(o_2), resp(o_2), \ldots$ and so on.

We choose $lp(o_i) := \max\{inv(o_i), lp(o_{i-1})\} + \frac{T}{n}, i \in \{1, 2, \ldots, n\}$, where we define $lp(o_0) := 0$.

It is clear that the sequence $(lp(o_n))$ is strictly increasing and that $\forall i, lp(o_i) \geq inv(o_i)$. We still need to show that $lp(o_i) \leq resp(o_i)$ before we can say that $lp(o_i)$ are linearization point.

$lp(o_i) - \frac{T}{n}$ is either $inv(o_i)$ or $lp(o_{i-1})$.

- If $inv(o_i)$, then by our choice of T, $inv(o_i) + \frac{T}{n} < inv(o_i) + T \leq resp(o_i)$
- If $lp(o_{i-1})$, it suffices to show $lp(o_{i-1}) + \frac{T}{n} \leq resp(o_i)$. We can again consider cases, whether $lp(o_{i-1}) - \frac{T}{n}$ is $inv(o_{i-1})$ or $lp(o_{i-2})$, and if it's the first case, we again have $inv(o_{i-1}) + 2\frac{T}{n} \leq inv(o_{i-1}) + T \leq resp(o_i)$. So by repeating this process, it suffices to show that $lp(o_1) + (i-1)\frac{T}{n} = inv(o_1) + i\frac{T}{n} \leq resp(o_i)$, which is true.
  To be even more rigorous, for fixed i, we choose the largest $j \leq i$ for which $lp(o_j) - \frac{T}{n} = inv(o_j)$, so that $lp(o_i) = lp(o_j) + (i-j)\frac{T}{n} = inv(o_j) + (i-j+1)\frac{T}{n} \leq inv(o_j) + T \leq resp(o_i)$

Hence, lp is a linearization point choice function. Furthermore, the order of linearization points corresponds to the order imposed by S, so that the execution must be legal (since S is legal). $\square$

Register ADT

A data type supporting Read(), Write(x).

Proposition An atomic register is regular.

Proof outline.
Consider cases:
Assume that the register is atomic.
Case 1: The read is not overlapping with any write.

- Show that there does not exist a linearization such that the read reads from a write that is after the read.
- Show that there does not exist a linearization such that the read reads from a write before the read that is not one of the most recent Formally, show that if w_1, w_2 are 2 write operations where $w_1, w_2 <_H r$ and $w_1 <_H w_2$, then read r does not read from w_1.
- Show that it is legal for the read to read from any of the most recent writes, i.e., we pick a particular most recent write w and show that there exists a linearization where the value read is due to w.

Case 2: The read is overlapping with at least one write.
Use the same ideas employed in case 1.

Models and Clocks

The difference between this chapter and the previous (consistency conditions) is that the previous chapter considers a parallel system, where there is indeed a single physical clock (the system clock), but this chapter considers a distributed system, where systems can be geographically separated, so there can be no single clock.

The other difference (a consequence of the first) is that a total ordering of *events* (but not necessarily operations) is possible in the parallel case, but not so for distributed systems. Due to it being impossible or extremely difficult to synchronize clocks, slight differences in clocks lead to events very close together being not possible to order. For e.g., suppose we have 2 geographically separated systems S1 and S2. An event e1 occurs at time t based on S1's physical clock, and an event s2 occurs at time $t + \delta t$ based on S2's physical clock. If δt is sufficiently small (perhaps smaller than the relative inaccuracies between S1 and S2's clocks), can we truly say e1 occured before e2 based on the physical clock? The answer is no.

Lemma An interesting lemma from the textbook. Suppose $\rightarrow$ is a partial order (non-total), and $<$ is a total order. Suppose also that $\forall a, b, a \rightarrow b \implies a < b$. Then it cannot be true that $a < b \implies a \rightarrow b$.

The reason is that if $a < b \implies a \rightarrow b$ were to hold, $\rightarrow$ would then be a total order.

Vector clock

There is a difference between the lecture's version of the vector clock and the textbook's version.
They agree on these properties:

- For an intra-process event, `clock[pid]++`, where `pid` is the index associated with the process
- When receiving a message, the following is executed

```
class VectorClock {
    const clock: Array<number>; // Vector of length n
    const pid: number;
    function receive(senderClock: Array<number>) {
        for (let i = 0; i < n; i++) { // component-wise maximum
            clock[i] = Math.max(clock[i], senderClock[i]);
        }
        clock[i]++;
    }
}
```

However, there is a difference when sending messages.

- The lecture's vector clock does `clock[pid]++` prior to sending the message. So the act of sending the message is similar to a regular intra-process event.
- Whereas the textbook's vector clock does `clock[pid]++` *after* sending the message.
  The same can be said about the matrix clock algorithm.

A mathematical description of the vector clock protocol

Let T_e be the vector clock right after the event e.
Local computation: Suppose e_1, e_2 are on the same process i, and e_2 happened right after e_1, so e_2 is a local computation.
Then,

- $T_{e_2}[i] \leftarrow T_{e_1}[i] + 1$
- $\forall j \neq i, T_{e_2}[j] \leftarrow T_{e_1}[j]$

Send-receive: Suppose s is a send event by process i and t is a receive event by process j. Then,

- $T_t[j] \leftarrow \max\{T_s[j], T_{prev}[j]\} + 1$
- $\forall k \neq j, T_t[k] \leftarrow \max\{T_s[k], T_{prev}[k]\}$

Claim. For the vector clock protocol, given events s, t, $s \rightarrow t \iff T_s < T_t$, where T is the vector clock logical time at the point of the event.

($\implies$) Consider cases.

1. s and t are both events on a single process i, and s is a computation event right before t. Then by definition of the vector clock protocol, $\forall j \neq i, T_s[j] = T_t[j]$ and $T_s[i] < T_s[i] + 1 = T_t[i]$, so that $T_s < T_t$.
2. s and t are events on different processes i and j respectively, and s is send event that corresponds to t, a receive event. By definition of the vector clock protocol, taking the component-wise maximum of T_s and the predecessor of T_t and incrementing the jth component of the predecessor of T_t means that $T_s < T_t$.
3. s happened before t via a chain of cases 1 and 2. Since $<$ between T is transitive, and we already showed that $s \rightarrow t \implies T_s < T_t$ for cases 1 and 2, $s \rightarrow t \implies T_s < T_t$ for case 3 as well.

($\impliedby$) Suppose s is an event on process i and t is an event on process j. Suppose that $T_s < T_t$. In particular, $T_s \leq T_t$ and $T_s[i] \leq T_t[i]$. We observe that the ith component of a vector clock time T can only be incremented by process i. Hence, if we were to search through all ancestors of event t via the happened before relation, we must be able to find s. Because otherwise, then any ancestor event belonging to process i will have its ith component strictly smaller than $T_s[i]$, and so $T_t[i] < T_s[i]$, which is a contradiction.

Matrix clock

A mathematical description of the matrix clock protocol

Let T_e be the matrix clock right after the event e.
Local computation: Suppose e_1, e_2 are on the same process i, and e_2 happened right after e_1, so e_2 is a local computation.
Then,

- $T_{e_2}[i, i] \leftarrow T_{e_1}[i, i] + 1$
- $\forall (j, k) \neq (i, i), T_{e_2}[j, k] \leftarrow T_{e_1}[j, k]$

Send-receive: Suppose s is a send event by process i and t is a receive event by process j. Then,

- $T_t[j, j] \leftarrow \max\{T_s[i, j], T_{prev}[j, j]\} + 1$
- $\forall k \neq j, T_t[j, k] \leftarrow \max\{T_s[i, k], T_{prev}[j, k]\}$

- $\forall k \neq j, \forall l, T_t[k,l] \leftarrow \max\{T_s[k,l], T_{prev}[k,l]\}$

Global snapshots

Global snapshots give a *possible* state in the past rather than what may actually have happened. Despite this, they are still useful in the following areas.

- Debugging distributed systems. Firstly it is not feasible to stop all remote systems at the same time. Next, given a global snapshot, even if the snapshot may not actually have happened, it can still be used to check the correctness of a distributed algorithm. For e.g. a distributed algorithm may give certain state guarantees that no global snapshot should violate.
- Checkpointing in long running computations. E.g. when running a distributed simulation, systems may crash. By rolling back to a previous snapshot, the computation can proceed from there. The interesting thing is that we might be rolling back to a snapshot that may not actually have happened (think multiverse!), but assuming that the distributed algorithm is coded correctly, the output should still be correct regardless of the global snapshot we roll back to.

Definition Given a set S of events, a global snapshot $G \subset S$ is such that $\forall e \in G$, any ancestor of e in the same process is also in G.

As a consequence, a global snapshot respects point 1 of the happened before relation.

Definition A consistent global snapshot is one that respects send-receive order.

i.e. such a snapshot respects point 2 of the happened before relation.

By inductive argument, a consistent global snapshot respects point 3 (transitivity) of the happened before relation.

Proposition Given a set of distributed processes, consider a single process P with events $e_1, e_2, \ldots$. Then, for any $m \in \mathbb{N}$, we can find an upper bound as well as a lower bound to a consistent global snapshot containing $e_1, \ldots, e_m$ and does not contain $e_{m+1}, e_{m+2} \ldots$ - $(\star)$.

The lower bound L is simply all ancestors of e_m with respect to the happened before relation. To see this we can show that

- Any consistent global snapshot G satisfying $(\star)$ must contain L, so that L is a subset of the lower bound. This is trivial to show since G respect the happened before order.
- L is itself a consistent global snapshot satisfying $(\star)$. To show this, consider any event $f \in L$ and suppose f' happened before f, i.e. $f' \rightarrow f$. Then, we have $f' \rightarrow f \rightarrow e_m$, so that $f' \rightarrow e_m$, so that f' is an ancestor of e_m. By our construction, $f' \in L$.
  - It is clear that L contains $e_1, \ldots, e_m$ but not $e_{m+1}, e_{m+2}, \ldots$.

The upper bound U is the setwise complement of e_{m+1} and all its descendants. Formally, $U := \{f : e_{m+1} \to f\}^c$. To see this we show that

- Any consistent global snapshot G satisfying $(\star)$ must satisfy $G \subseteq U$. Equivalently, $U^c \subseteq G^c$. Clearly, if $f \in U^c$, then $e_{m+1} \to f$, so $f \notin G$, so $f \in G^c$.
- U is itself a consistent global snapshot satisfying $(\star)$. We consider any element $f \in U$ and suppose that $f' \to f$. We note that $e_{m+1} \not\to f$. Since $f' \to f$, $e_{m+1} \not\to f'$, and by construction, $f' \in U$.
    - It is also clear that U contains $e_1, \ldots, e_m$ but not $e_{m+1}, e_{m+2}, \ldots$.

We have thus established an upper and lower bound. $\square$

Chandy and Lamport's protocol

Here are some characteristics of the protocol. Suppose there are n distributed processes. Then,

- Each process will send out 1 Marker message to the other $n-1$ processes when it takes its local snapshot.
- Since each process sends $n-1$ Marker messages, there are $n(n-1)$ Marker messages associated with one global snapshot.
- It is not necessary for there to be exactly 1 process to initiate the protocol. For e.g., there may be more than 1 initiator. This can happen when more than 1 process tries to take a local snapshot before receiving any Marker messages.
- For a process, it can take its local snapshot either (1) before any Marker messages is received, (2) when the first Marker message is received.
- The local snapshot is instantaneous but processes would want to capture messages in transit. Because of this, each process will record down the messages received between its local snapshot and any Marker message that is received.
    - For example, consider process 1, and it takes a local snapshot at time t_1, and receives Marker messages from process 2, 3, ..., n in this order. We let the time of reception of the ith Marker message be $t_i, i = 2, 3, \ldots, n$. Then process 1 will record down the following info: messages in (t_1, t_2), messages in (t_1, t_3), ... messages in (t_1, t_n).
    - By examining the messages in (t_1, t_5) for example, allows us to determine what messages were in transit between process 1 taking its local snapshot and receiving the Marker message from process 5.
    - In general, all on the fly messages from process i to process j will be captured in the time between process j's snapshot and the time at which process j receives the Marker message from process i.
    - This is reason why there are $n(n-1)$ Marker messages. If we do not care about messages on the fly, only the initators need to send out Marker messages, so there would only be $k(n-1)$ Marker messages, where k is the number of initiators.

Proposed Definition On the fly message/Message in transit: Let s, t be the send, receive events associated with a message m. m is considered to be on the fly with respect to a certain global snapshot G if G contains s but not t.

Message Ordering

In the previous 2 chapters, we only concerned ourselves with the happened before order (for e.g. a consistent global snapshot respects the happened before order). In this chapter, more orders are introduced.

Important Remark In this chapter, the "receive" event is really the "deliver" event. Suppose process i sends a message to process j, j may receive the message without delivering it first. In this case, we do not consider there to be any "receive" event, until the actual delivery occurs.
This means that when we discuss happened before ordering of send-receive events s, t, where s is the send event by i and t is the receive event by j, t is more accurately the delivery event. In the proof of correctness of the causal ordering protocol, we make use of r to denote the plausible receive event, this is when r is received but not necessarily delivered. (Yes, this is confusing.) So when r is put on hold and not delivered, there is then no concrete event.
In the context of Skeen's algorithm however, we do need to care about the original receive event, since logical clocks are incremented immediately on message reception
. (Yes, this is very confusing).

Really, the source of this confusion is because we are attempting to mix message reception with message delivery. But they are distinct.

Causal order

Requirement: Given send events s_1, s_2, if $s_1 \to s_2$ (happened before), and r_1, r_2 are on the same process, then $r_1 \to r_2$.
The idea is that s_1 could have caused s_2, so r_1 should be received before r_2.

Causal ordering protocol

We define the following notation of a causal ordering protocol matrix.
For any matrix M, let $M[i, j]$ be the (i, j)-th entry.
For a send event s,

$$M_{i,s}$$

refers to the matrix M of a process P_i at a send event s. i.e. P_i is the sender.

For a *plausible* receive event r,

$$M_{i,r}, T_{i,r}$$

refers to the matrix M, T of a process P_i whose has received but not necessarily delivered a message. T comes from the sender and is piggybacked on the message.

We can let $M^*_{i,r} = \mathrm{ComponentWiseMaximum}(M, T)$ denote the matrix *after* the receive event r actually occurs, i.e. the message is delivered.
Otherwise if message delivery is delayed, the "latest" matrix of P_i remains as $M_{i,r}$. (Technically in this case r doesn't exist at this point in time and is delayed, but we will still denote r as the hypothetical receive event that could have occurred at this point in time.)

Fact For a send event s from P_i to P_j and its corresponding (plausible) receive event r, $M_{i,s} = T_{j,r}$.

When process P_j receives a message from P_i with associated matrix $T_{j,r}$, the message is delivered iff

$$\begin{cases} M_{j,r}[l,j] \geq T_{j,r}[l,j], & l \neq i \\ M_{j,r}[i,j] = T_{j,r}[i,j] - 1 \end{cases}$$

The idea is that $\forall l \neq i, M_{j,r}[l,j] \geq T_{j,r}[l,j]$ represents that P_j has received all messages sent to it by other processes P_l, whose send events happened before the send event by P_i.
$M_{j,r}[i,j] = T_{j,r}[i,j] - 1$ represents that P_j has received all the first $T_{j,r}[i,j] - 1$ messages (assuming matrices start from the zero matrix) and the $T_{j,r}[i,j]$-th message is the most recent undelivered message from P_i.
With both conditions fulfilled, it suggests that this message can be delivered without violating causal order.

Claim The protocol is correct, in that

1. Causal order is preserved.
2. All sent messages can eventually be delivered.

We first prove claim (1), with respect to 2 send events s_1, s_2, and their corresponding receive events r_1, r_2, where the receive events take place on the same process P_j. Let $s_1 \to s_2$.

Case 1: s_1 and s_2 take place on different processes. WLOG, by possibly renumbering the processes, let s_1 take place on process P_1 and s_2 take place on process P_2.

Since $s_1 \to s_2$, and along a path in the happened before graph, we can state the following facts.

- $M_{1,s_1}[1,j] \leq M_{2,s_2}[1,j]$ since component-wise maximums are taken for receive events, and for send events, the matrix is strictly increasing, so that happened before order implies "monotonically increasing" matrices.
- $M_{1,s_1}[2,j] < M_{2,s_2}[2,j]$. This is due to P_2 incrementing the $(2,j)$-th entry during the send event s_2.

Using the **Fact** above, we can translate this into T equivalents. That is,

- $T_{j,r_1}[1,j] \leq T_{j,r_2}[1,j]$
- $T_{j,r_1}[2,j] < T_{j,r_2}[2,j]$

Now, we assume that r_2 happens before r_1, then consider $M^*_{j,r_2} = \text{ComponentWiseMaximum}(M_{j,r_2}, T_{j,r_2})$. This implies that

$$M^*_{j,r_2}[1,j] \geq T_{j,r_2}[1,j] \geq T_{j,r_1}[1,j] > T_{j,r_1}[1,j] - 1$$

which implies that r_1 can never occur based on the protocol.

Hence, we have proven that if r_2 happens before r_1, then r_1 can never occur. Assuming claim (2) is true, then r_1 must eventually occur, so we have a contradiction.

Case 2: $s1, s2$ take place on the same process. WLOG, let this process be P_1.
Then we must have $M_{1,s_1}[1,j] < M_{1,s_2}[1,j]$ since P_1 increments the $(1,j)$-th entry at least once due to s_2. This translates to $T_{j,r_1}[1,j] < T_{j,r_2}[1,j]$.

Similarly as in case 1, we assume that r_2 happens first, and we get the inequality

$$M^*_{j,r_2}[1,j] \geq T_{j,r_2}[1,j] > T_{j,r_1}[1,j] > T_{j,r_1}[1,j] - 1$$

which implies that r_1 can never occur.
We obtain a similar contradiction assuming claim (2) is true.

Notice that claim (1) depends on claim (2) to be true. We now prove claim (2).
For a receiver P_j, let $\mathbf{x} = M[\cdot, j]$ be the jth column of the most recent matrix. Notice there is no subscript here as we do not know (nor do we care) which plausible read event to reference.
Let $S_1 \neq \emptyset$ be the set of undelivered messages. For each sender P_i who has sent at least 1 message still in S_1, there must be a message whose matrix $T[i,j] = x_i + 1$. We give the following argument. Let T be the matrix of the message sent by P_i whose (i,j)-th component is the smallest.

- We cannot have $T[i,j] \leq x_i$ as this implies that this message is the $T[i,j]$-th message sent by P_i and P_j has delivered exactly the first x_i messages, which includes the $T[i,j]$-th message.
- We cannot have $T[i,j] > x_i + 1$ indefinitely. The reason is that we assume that all sent messages eventually arrive. Since the $T[i,j]$-th message has been sent, so must the $x_i + 1$-th message, which must be eventually receive. So eventually, we can pick the T where $T[i,j] = x_i + 1$.

Following the lecture slides, we call this a successor message.
Let $S_2 \subseteq S_1$ be the set of successor messages.

- $S_2 \neq \emptyset$ since for each $i \neq j$, we can find one such successor message.
- S_2 has at most one message from any sender since $x_i + 1$ is one number, and all messages from the same sender have distinct $T[i,j]$. (Each send event increments $T[i,j]$ by one.)

Hence $1 \leq |S_2| \leq n$.

We can then let $S_3 \subseteq S_2$ be the messages whose corresponding send events did not happen after any other send events in S_2, i.e., $\forall s_3 \in S_3, s_2 \in S_2, s_2 \not\to s_3$. Following the lecture slides, we call S_3 the set of top successor messages.

Claim: Any top successor message can be delivered. By proving this, we can prove that all sent messages can eventually be delivered by mathematical induction on the order of the message with respect to its sender process.

Consider a particular message $m \in S_3$ and its corresponding matrix T. We assume that the message is sent by process P_1. For m to be deliverable, we recall the conditions:

$$\begin{cases} M[l,j] \geq T[l,j], & l \neq 1 \\ M[1,j] = T[1,j] - 1 \end{cases}$$

Since m is a successor message, we have $T[1,j] = x_1 + 1 = M[1,j] + 1$. It then suffices to show that $\forall l \neq i, M[l,j] \geq T[l,j]$. Suppose this is not true for $l = 2$ (WLOG), i.e. $M[2,j] < T[2,j]$. This suggests that process P_2 has a send event that happened before P_1's send event corresponding to m. Indeed, we consider the $(M[2,j]+1)$-th message m_1 sent by P_2 to P_j. Note: $M[2,j] < M[2,j] + 1 \leq T[2,j]$. Furthermore,

- The send event by P_2 must happen before the send event by P_1.
- m_1 is a not yet delivered successor message, as $M[2,j]$ is not yet updated. (Otherwise, a component-wise maximum would result in $M[2,j]$ getting increased.)

These 2 points imply that m is not a top successor message, which is a contradiction.

We have thus proven that any top successor message can be delivered. $\square$

Remark While the proof above is relatively rigorous, we implicitly assumed the following proposition: For an event e on process P_1 with associated $M[2,j]$ the send events by P_2 with associated T where $T[2,j] \leq M[2,j]$ all happen before e. Furthermore, the send events by P_2 with associated T where $T[2,j] > M[2,j]$ do not happen before e.

Broadcast Messages

This is similar to the regular non-broadcast case, with the exception that a process can send a message to itself.

Causal ordering

This is fine (the fact that a process can send messages to itself), this just means that for a matrix M, the entries along the diagonal, $M[i,i]$ can be greater than zero.

Total ordering

Total ordering only applies to broadcast messages. This makes sense since all processes have to agree on a particular total ordering of messages, and for all processes to be aware of a message, the message should be a broadcast.

It is precisely the requirement for a process to send messages to itself that results in the possiblity of having total ordering but not causal ordering.
For example, consider the totally ordered sequence of messages in a broadcast group with nodes v_1, v_2.

- `v_1.send(m_1, to = v_1)` , `v_1.send(m_2, to = v_1)` , `v_1.send(m_1, to = v_2)` , `v_1.send(m_2, to = v_2)` , `v_1.deliver(m_2)` , `v_2.deliver(m_2)`
- `v_2.deliver(m_2)` , `v_2.deliver(m_1)`

Here, v_1, v_2 both agree on delivering m_2, then m_1, even though the send event for m_1 happened before the send event for m_2. Hence, we have total ordering of the message delivery but not FIFO order. Since causal ordering implies FIFO ordering, we do not have causal ordering.

Skeen's algorithm for total order broadcast

We examine the places where a process's logical clock is incremented.
Note that the logical clock protocol is being run like normal.
Suppose process P1 broadcasts a message m to the other processes P2, P3.

- P1: P1.clock += 1, send(m)
- P2: buffer.receive(m), P2.clock = max{m.clock, P2.clock} + 1; similarly for P3
- P2: P2.clock += 1, send(ACK); similarly for P3
- P1: receive(P2.ACK), P1.clock = max{P2.ACK.clock, P1.clock} + 1; similarly for the ACK from P3
- P1: P1.clock += 1, notify(max_clock_received)
- P2: receive(message_number), P2.assign(m, message_number), P2.clock = max{P1.notify.clock, P2.clock} + 1; similarly for P3

Since message delivery does not count as an event, it will not change the logical clock value.

We restate a property of Skeen's algorithm from lecture.
Proposition Suppose message m_1 on (receiver) process P_j has received its message number, and another message m_2 arrives on P_j. Then the eventual message number of m_2 must be strictly greater than that of m_1.

This property holds as a result of the logical clock protocol, and the fact that ACK-ing a message (m_2 in this case) counts as a send event, and so increments the logical clock of P_j. Using the above discussion P1.notify.clock is an upper bound for the message number, and P2.clock = max{P1.notify.clock, P2.clock} + 1 means that P2.clock will be strictly greater than the message number of m_1 even after receiving the notification. So when P2 sends the ACK for m_2, its logical clock will be even larger. $\square$

Remark This detail is not mentioned in lecture, nor in textbook. It seems that Skeen's Algorithm does not prevent duplicate sequence numbers from different processes.
Consider 2 processes, P1, P2

- P1 send to P2
- P2 send to P1
- P1 receive, P1.clock++
- P2.receive, P2.clock++
- P1 sends ACK 1 to P2
- P2 sends ACK 1 to P1
- Both messages have sequence number 1

However, this may not be an issue, as tiebreaking can be done via the process index. e.g. (seq num = 1, sender = P1) < (seq num = 1, sender = P2).

Homework notes

Some formal definitions used in the homework questions.
For events e, f on the same process, $e \prec f$ means that e is before f in process order. $\to$ is the usual happened before order.
Causal order

$$s_1 \to s_2 \implies \neg(r_2 \prec r_1)$$

FIFO order

$$s_1 \prec s_2 \implies \neg(r_2 \prec r_1)$$

Think TCP in computer networking, point to point communication can ensure FIFO delivery by using sequence numbers.

Since $s_1 \prec s_2 \implies s_1 \to s_2$, causal order implies FIFO order. We used the FIFO delivery during global snapshots. FIFO order delivery is needed to ensure that the Marker message is delivered before other messages from the same sender (sent after the Marker message).

Leader Election

Proposition On an anonymous ring with deterministic algorithms, leader election is impossible.
Since the ring is anonymous, all nodes have the same state. Furthermore, assuming the algorithm runs at the same speed on each node, so all nodes always have the same state. Hence, either none of the nodes are the leader, or all of the nodes are the leader (which is not correct since there can be only 1 leader). $\square$

Chang-Roberts Algorithm

Let n be the number of nodes on the ring.
Best case performance: n
Proof Only the node with the largest id can be the leader, so at the very least this node has to send its message all the way around the ring, which incurs n steps. Only when the message travels around the ring and returns to the node with the largest id, does this node know that it is the leader. Hence, we have the tight lower n messages. $\square$

Note that if all nodes manage to send at least 1 message, then the best case performance would be $n + n - 1 = 2n - 1$.

Worst case performance: $\frac{n(n+1)}{2}$
Proof Consider the node v_k with rank k in the ring. By rank, I mean the node with the k-th smallest number. There is exactly one node of each rank $k \in \{1, 2, \dots, n\}$.
There are exactly $k - 1$ nodes smaller than v_k, so the message from n_k will be retransmitted at most $k - 1$ times. Accounting from v_k's own transmission, the message will be transmitted at most k times.
Hence, the upper bound of messages transmitted is $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. We now give a cyclic permutation of the nodes that achieves this upper bound.

$$v_n \to v_{n-1} \to v_{n-2} \to \cdots \to v_1 \to v_n$$

i.e. the cyclic permutation is in descending order before wrapping around. (the arrow points in the direction of message transmission)
Then we see that the message from a node v_k of rank k will be able to pass through all nodes $v_i, i < k$, and so the upper bound is tight. $\square$

Average case performance: $O(n \log n)$ (Claim)
Proof Unlike the lecture notes, we will give an exact calculation, which doesn't make use of bounds.

The sample space is over the $(n-1)!$ distinct cyclic permutations of $\{1, 2, \dots, n\}$.
Define random variable X_k similarly as in lecture, which is the number of messages originating from v_k (the node of rank k) that can be transmitted. More simply, $X_k - 1$ is the largest number of consecutive nodes with rank $< k$. The total number of messages T is given by $T = \sum_{1 \le k \le n} X_k$, so by linearity of expectation we get average $T(n) = E[T] = \sum_{1 \le k \le n} E[X_k]$.

We derive some useful probabilities.

- $\forall i \le k, P(X_k \ge i) = \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}}$. The reason is that the event $\{X_k \ge i\}$ is equivalent to the event where the $i - 1$ nodes right after v_k come from the $k - 1$ nodes with rank smaller than k. $(n - 1)$ is the number of the nodes other than v_k.
- Clearly, $i > k \implies P(X_k \ge i) = 0$
- $\forall i \le k, P(X_k = i) = P(X_k \ge i) \cdot \frac{n-k}{n-i}$. The reason that following the $i - 1$ nodes after v_k, the i-th node after v_k must have rank $> k$, and there are $n - k$ such nodes. However, each of the remaining

$n - i$ nodes are equally likely to be this i-th node, so the probability that any of them is the i-th node is $\frac{1}{n-i}$.
We can verify the last result by considering the following computation. If $1 \le i \le k$, then

$$\begin{aligned} P(X_k = i) &= P(X_k \ge i) - P(X_k \ge i + i) \\ &= \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}} - \frac{(k-1)^{\underline{i}}}{(n-1)^{\underline{i}}} \\ &= \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}}(1 - \frac{k-i}{n-i}) \\ &= \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}} \frac{n-k}{n-i} \end{aligned} \tag{1}$$

which shows that our first and last result agree.

We note that X_k is a non-negative, integer valued random variable. This means that

$$\begin{aligned} E[X_k] &= \sum_{i \in \mathbb{Z}^+} P(X_k \ge i) \\ &= \sum_{1 \le i \le k} \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}} \end{aligned} \tag{2}$$

This isn't so easy to evaluate. Now we try interchanging summations.

$$\begin{aligned}\sum_{1\le k\le n} E[X_k] &= \sum_{1\le k\le n}\sum_{1\le i\le k} \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}} \\ &= \sum_{1\le i\le k\le n} \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}} \\ &= \sum_{1\le i\le n}\sum_{i\le k\le n} \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-i}}} \\ &= \sum_{1\le i\le n} \frac{(i-1)!}{(n-1)^{\underline{i-i}}} \sum_{i\le k\le n} \frac{(k-1)^{\underline{i-1}}}{(i-1)!} \\ &= \sum_{1\le i\le n} \frac{(i-1)!}{(n-1)^{\underline{i-i}}} \sum_{i\le k\le n} \binom{k-1}{i-1} \\ &= \sum_{1\le i\le n} \frac{(i-1)!}{(n-1)^{\underline{i-i}}} \binom{n}{i} \\ &= \sum_{1\le i\le n} \frac{(i-1)!}{(n-1)^{\underline{i-i}}} \frac{n^{\underline{i}}}{i!} \\ &= \sum_{1\le i\le n} \frac{n}{i} = nH_n = n\Theta(\log n) = \Theta(n\log n)\end{aligned} \tag{3}$$

Above, we make use of a combinatorial identity that states that $\binom{n+1}{r+1} = \sum_{r\le k\le n}\binom{k}{r}$.
Anyway, we have an exact result average $T(n) = nH_n$. $\square$

We also give a a more rigorous justification for the lecture's derivation of the big-O bound for average number of messages.
As derived above, $P(X_k \ge i) = \frac{(k-1)^{\underline{i-1}}}{(n-1)^{\underline{i-1}}} \le (\frac{k-1}{n-1})^{i-1} = (1-\frac{n-k}{n-1})^{i-1} = (1-p)^{i-1} = P(Y\ge i)$ where $p = \frac{n-k}{n-1}, Y \sim Geom(p)$.

As a result, $E[X_k] \le E[Y] = \frac{1}{p} = \frac{n-1}{n-k}$, and $\sum_{1\le k\le n} E[X_k] = E[X_n] + \sum_{1\le k\le n} \le E[X_n] + (n-1)\sum_{1\le k\le n-1}\frac{1}{n-k} = n + (n-1)H_{n-1} = O(n\log n)$. $\square$

Lemma Suppose X, Y are random variables such that $range(X) = \{1,2,\ldots,n\}, range(Y) = \mathbb{Z}^+$ and $\forall 1\le i\le n, P(X=i \mid X>i-1) \ge P(Y=i\mid Y>i-1)$.
Then $E[X]\le E[Y]$.

Proof Note that $E[X] = \sum_{i\ge1} P(X\ge i) = \sum_{1\le i\le n} P(X\ge i)$, $E[Y] = \sum_{1\le i\le n} P(Y\ge i) + \sum_{i>n} P(Y\ge i)$. It then suffices to show that $\forall 1\le i\le n, P(X\ge i)\le P(Y\ge i)$.

The base case holds since $P(X\ge 1) = 1 = P(Y\ge 1)$. The inductive hypothesis is that for $1\le i< n, P(X\ge i)\le P(Y\ge i)$.
We have

$$
\begin{aligned}
&P(X=i \mid X>i-1) \geq P(Y=i \mid Y>i-1) \implies \\
&\frac{P(X=i)}{P(X \geq i)} \geq \frac{P(Y=i)}{P(Y \geq i)} \implies \\
&\frac{P(X \geq i+1)}{P(X \geq i)} \leq \frac{P(Y \geq i+1)}{P(Y \geq i)} \implies \\
&P(X \geq i+1) \leq P(Y \geq i+1)
\end{aligned} \tag{4}
$$

and we are done. □

Distributed spanning tree

Algorithm.

1. Initialization: The initiator node sends out child requests to its neighbors.
2. All other nodes will execute the following protocol:
   1. Wait until at least one child request is received.
   2. Select the first child request, and discard the rest. NAK (non-acknowledgements) can be sent.
   3. Send child requests to neighbors.
   4. Wait for all recipients of child requests to reply.
   5. Acknowledge the sender of the selected child request as parent.
3. The algorithm terminates when the root node receives all replies from its neighbors.

Proposition The spanning tree construction algorithm will not result in a cycle.

Proof Make use of happened before order of receive events. Choose a particular node in the cycle and consider its parent. The child request send event of its parent must have happened before its receive. Keep repeating this argument until we have gone around the cycle. This implies that the happened before relation creates a cycle, which is not possible. (This can be viewed as an axiom.) □

Proposition The spanning tree construction algorithm results in a tree rooted at the initiator.

Proof

- We already know from above that the algo gives an acyclic graph.
- Each node (other than the initiator) will select exactly one parent, hence there will be $n-1$ edges in the resulting graph
- Hence, we have a acylic graph with $n-1$ edges, so it must be a tree.
- Suppose there is more than 1 root. Then, we can argue that we can find a node with at least 2 outgoing edges, i.e. more than 1 parent. But this contradicts the algo, where a node can choose at most 1 parent. □

Discussion Benefits of the spanning tree with regard to leader election.
Let n be the number of nodes in a general connected graph.

If n is unknown, then the spanning tree protocol is needed to compute n.
If n is known, then when each node floods the graph with its identifier, it is possible to elect a leader. For e.g. a node, upon receiving $n-1$ messages, would be able to check if its identifier is the largest, and therefore decide whether it is the leader.

However, even if n is known, there are still benefits to the spanning tree algorithm as it removes the need to broadcast.

Other notes on the spanning tree

- It is actually not necessary to construct a spanning tree before doing a computation like counting the number of nodes. A similar procedure like the spanning tree construction can be used when directly doing a computation like counting nodes. The only difference between already having a spanning tree in place, and without, is that in the latter case, there may be messages sent along non-tree nodes which are met with NAKs.
- Order of events: We list down the following events.
  - (1): A node receives the first child request message. (In a direct computation without having constructed the spanning tree, we can view the computation request as a child request)
  - (2): A node sends child request messages to its neighbors after (1).
  - (3): A node replies ACK to its chosen parent (who corresponds to the first child request message). (In the case of a direct computation, this would be replying the computation result to its "parent".)
  - (4): A node replies NAK to neighbors who also sent it child request messages, but are not the chosen parent. (In the case of a direct computation, this would still be a NAK.)
  - (5): A node has received all replies to its child request messages, be it ACK or NAK.
  - Then, we must have the following orderings:
    - (1) before everything else. Obvious.
    - (5) -> (3). This is important as replying an ACK in the case of (3) can be used as an indication that the node has "connected" successfully (in the event of ACK) or unsuccessfully (in the event of NAK) with all its neighbors.
      For e.g. in spanning tree construction, when the root receives ACK/NAK from its children, it knows that the subtree at its children have been fully formed.
      For e.g. in counting nodes, it is simply impossible for a non-leaf node to give the correct reply in (3) without having completed (5).
    - (4) whenever possible. There is no reason to delay NAKs.

Distributed Consensus

https://cs.stackexchange.com/questions/105716/what-is-the-difference-between-consensus-and-leader-election-problems
Some pointers from the link

- Leader election is a special case of consensus.

- They are equivalent when there are no failures.
  - Given a consensus algorithm A, a leader election algorithm B can be formed where each node chooses themselves and after applying A, the decision will be the leader.
  - Given a leader election algorithm B, a consensus algorithm A can be formed by choosing a leader via B, then letting the leader make the decision.
- However, there are certain failure models underwhich they are not equivalent. Sometimes, consensus can be solved, but not election.
  - The difference is due to the nature of a failure in a leader election. If the leader, denoted as L, fails, even if all other working nodes agree on L, this may be considered consensus, but in terms of leader election, this is a "wrong" answer. A crashed node cannot be a leader.

Differences in various models

- Failure model
  - Node failure (Crashed nodes / Byzantine failure)
  - Channel failure (Message loss)
- Timing (Synchronous / Asynchronous)

Desirable properties of a consensus algorithm

- Termination
- Agreement
- Validity: On cs.stackexchange, I have seen a definition of validity that says that when there are nodes $v_1, \ldots, v_n$ each with input $i_1, \ldots, i_n$ (possibly exists j, k for which $i_j \neq i_k$) making a decision d, d must be one of $i_1, \ldots, i_n$. However, in CS4321, validity only comes into effect when $i_1 = i_2 = \cdots = i_n$.

Model 1

- Node failure
- Channel reliable
- Synchronous

The initial input of each node v is represented as a set S_v.
Intuition: A sufficient condition to achieve consensus is for the available information S_v of each node to be shared, and one way to model information sharing is to take set union.
It is possible to differentiate between the sets of different nodes by simply labelling the set with its associated node.
For e.g. initially, node v_i has $\{(i, D_i)\}$, where D_i is the set of data of v_i and the i is meant to identify this tuple as belonging to v_i. Then the final union might look something like $\{(1, D_1), (2, D_2), \ldots\}$. Intuitively, this captures all possible information.

Model 2

- Node reliable
- Channel failure
- Synchronous

Proposition Termination, Agreement and Validity cannot be achieved by any deterministic protocol.

Proof Suppose there exists an algorithm A that achieves this. We transcribe the lecture's proof for 2 communicating nodes.
Notice that there is only 1 decision column due to agreement. Furthermore, we let the communication channel drop all messages, so that an execution of a node can *only* be distinguished by its input. In other words, as long as the input to a node does not change, by determinism of A, the decision of the node must remain the same.

Input1	Input2	Decision
0	0	0(validity)
1	0	0(node 2 indistinguishable)
1	1	0(node 1 indistinguishable)

The final row of the table shows a contradiction as it fails validity. The inputs to both nodes are 1, yet the decision is $0 \neq 1$.

The key idea of this proof is to change the input to one node at a time, since the input to the other node doesn't change, the decision cannot change. This idea is easily generalizable to the case of n communicating nodes, $n \geq 2$ by flipping each 0 input one at a time until all inputs are 1s. $\square$

Proposition Under weakened validity requirements as described in lecture, it is still not possible to achieve Termination, Agreement and weakened validity.

Proof While the lecture gives a proof for the case of 2 nodes, we will generalize this to n communicating nodes.
We consider the set of all messages m sent. Each m has an associated send and receive event, which we will denote as $send(m), recv(m)$ respectively. For an event e, denote $node(e)$ as its node.
Definition: A "last" message m is one for which $recv(m)$ does not happen before any *successful* send event on $node(recv(m))$.
It may be possible to make this definition more lax, however, for the purposes of this proof, it is not necessary to consider such relaxations, and the more restrictive definition also makes proving easier.

Part 1: Prove that there exists an execution in which all messages are lost, yet the decision is unchanged.

If the set of received messages (i.e. messages that are not lost by the channel) is empty, we can proceed on to the next part of the proof.

Otherwise, assume that the set M of received messages is non-empty. We claim that amongst this set, there exists at least one "last message". Clearly, for any $m \in M$, $send(m) \to recv(m)$. We can keep tracing $send(m_1) \to recv(m_1) \to send(m_2) \to recv(m_2) \to \ldots$ or $send(m_1) \to recv(m_1) \to recv(m_2) \to \ldots$. The idea is that for a message m_1, if m_1 is not a "last message", then $recv(m_1)$ happens before some event associated with some message m_2. This is either a send or a receive event.
Since there are finitely many messages, and no messages can repeat since $\to$ is a partial order, we eventually reach a receive event of a "last message".
Now that we have found the last message m_l, we claim that the same inputs, together with a set of messages $M \setminus \{m_l\}$ is indistinguishable for $node(send(m_l))$. The idea that any events that might be caused by the reception of m_l by $node(recv(m_l))$ must be completely localized to $node(recv(m_l))$ as it does not happen before any other send-receive event. In other words, for all other nodes $\neq node(recv(m_l))$, they cannot tell the difference between whether m_l was successfully received or whether m_l was dropped by the communication channel. In particular, this is also indistinguishable to $node(send(m_l))$. Note that it may be *plausible* that the dropping of m_l causes the intended receipient of m_l to generate some other events, some of which are send events, but we can simply drop all of those as well. In other words, we **guarantee** via the dropping mechanism that the set of received messages is indeed $M \setminus \{m_l\}$, nothing more.
Due to indistinguishability, the decision by all nodes other than the intended recipient of m_l are the same as before. And by the agreement property of a correct consensus algorithm, the intended recipient of m_l is *forced* to agree with the others, so its decision remains the same as well.
Thus, we have managed to reduce M to $M \setminus \{m_l\}$ whilst keeping the decision the same.
By repeating this process, (since there is always a "last message" in an nonempty set M), we can reduce M to the empty set.

Part 2: Using a similar technique to the previous proof of impossibility, show a contradiction.
We show a proof for 2 nodes

Input1	Input2	Decision
1	1	1(validity)
0	1	1(node 2 indistinguishable)
0	0	1(node 1 indistinguishable)

We have a contradiction, as the weakened validity requirements state that if both inputs are 0, the nodes must decide with 0.
Note that we started with 1s as inputs since we want to end with 0s as input. This is because both inputs 0 give a stronger requirement on the decision, that does not depend on the lossiness of the communication channel.
Again, this is easily generalizable to $n \geq 2$ nodes. $\square$

Allowing Limited Disagreement

Randomized algorithm A deterministic algorithm with a random bit string (possibly of infinite length) as input. If we consider a Turing machine, this is like having access to a tape with a randomized binary sequence.

An adversary is made aware of the probability distribution that the bitstring takes. But the adversary does not know the exact realization/outcome of the random variable beforehand.

For the next protocol, there exists an error probability ϵ such that the nodes disagree. Note that when agreement is not attained, it makes no sense to talk about validity.
However, with probability $1-\epsilon$, the nodes agree, furthermore, they attain the weakened notion of validity.

Some notes on the lecture's proof of the protocol for 2 nodes:
We give a more analytical proof of the lemma that at all times, $|L_1 - L_2| \leq 1$ and L_1, L_2 are monotonically increasing. We do so by induction. The base case is trivial, with $L_1 = L_2 = 0$. Next, suppose that L_1, L_2 satisfy the aforementioned properties. Denote the next values by L_1', L_2'.
Then, by definition of the protocol, $L_1' \in \{L_1, L_2+1\}, L_2' \in \{L_2, L_1+1\}$. Hence

$$(L_1', L_2') \in \{L_1, L_2+1\} \times \{L_2, L_1+1\} = \{(L_1, L_2), (L_1, L_1+1), (L_2+1, L_2), (L_2+1, L_1+1)\}$$

We can check that

- $|L_1 - L_2| \leq 1$ by inductive hypothesis
- $|L_1 - (L_1+1)| = 1$
- $|(L_2+1) - L_2| = 1$
- $|(L_2+1) - (L_1+1)| = |L_2 - L_1| \leq 1$

We have hence proven $|L_1' - L_2'| \leq 1$.

To see monotonicity, we can check for $L_1' \in \{L_1, L_2+1\}$ that

- $L_1 \geq L_1$
- $L_1 - 1 \leq L_2 \leq L_1 + 1$ so that $L_1 \leq L_2 + 1 \leq L_1 + 2$
  We can do similarly for L_2'. This completes the induction.

We transcribe the case analysis of agreement presented in lecture. Since {0, 1} are the only decisions, disagreement implies that one node decides 0 and the other decides 1.

Note that since there are r rounds, and in each round, L_1, L_2 each increases by at most 1, at the end, r is an upper bound for L_1, L_2.

Case 1: P1 sees P2's messages at least once, P2 does not see P1's message
Clearly, P2 decides 0, so P1 must decide 1. This implies

- $L_1 > 0$ since L_2 is monotonically increasing and $L_2 \geq 0$, so $L_1 \geq L_2 + 1 \geq 1 > 0$ as a result of L_1 having seen L_2 at least once.
- $L_1 \geq bar$

- $L_2 = 0$

Since $|L_1 - L_2| \leq 1$, the above facts say that

$$0 = L_2 < bar \leq L_1 \leq L_2 + 1 = 1$$

forcing $L_1 = 1$ and $bar = 1$. (Note that $bar > 0$, so we have the leftmost inequality for free.)

Case 2: P1 does not see P2's message, P2 sees P1's messages at least once
A similar argument gives

$$0 = L_1 < bar \leq L_2 \leq L_1 + 1 = 1$$

forcing $L_2 = 0$ and $bar = 1$.

Case 3: P1 sees P2's messages at least once, P2 sees P1's messages at least once.
Since both nodes have seen the other's messages at least once, the only condition left to check is if $L_i \geq bar, i = 1, 2$.
In either case, $\exists i \in \{1,2\}, j \in \{1,2\} \setminus \{i\}$, $L_j < bar \leq L_i \leq L_j + 1$ where P_i decides 1 and P_j decides 0.
This forces $L_i = L_j, bar = L_j + 1$.

Case 4: P1 doesn't see P2's message, P2 doesn't see P1's message
This results in both P1 and P2 deciding 0, so this does not result in disagreement.

As bar is unknown to the adversary, *conditional* on cases 1,2,3, bar can take exactly one value out of $\{1, 2, \ldots, r\}$, with probability $\frac{1}{r}$. Hence, conditional on cases 1,2,3 the probability of agreement is given by $1 - \frac{1}{r}$.
Furthermore, conditional on case 4, the probability of agreement is 1.
Since the adversary can always choose exactly which messages to drop, the "best" that the adversary can achieve is an error probability of $\frac{1}{r}$. $\square$

Validity When P1, P2 agree (with probability $1 - \frac{1}{r}$), their decision must be valid.

- Suppose P1, P2 both had input 0, then regardless of the channel conditions, they will decide 0, which is valid.
- Suppose P1, P2 both had input 1, and there is no message loss, then they must have $L_1 = L_2 = r$ at the end of r rounds, so that they will decide 1, which is valid.
- In all other cases, P1, P2 can agree on anything, hence valid.

Dicussion Consider some other ranges for bar. Note that the adversary may not know the (random) choice of bar, but the adversary knows the probability distribution of bar, i.e. *the adversary knows which range bar is uniformly chosen over*.

Range 1: $[0, r]$ instead of $[1, r]$.
Conditional on Case 1, 2, disagreement occurs with probability $\frac{2}{r+1}$. For example, the inequality in case 1 becomes

$$0 = L_2 \leq bar \leq L_1 \leq L_2 + 1 = 1$$

so $bar \in \{0, 1\}$ would work.
Conditional on Case 3, disagreement occurs with probability $\frac{1}{r+1}$

The best the adversary can do is to drop messages such that Case 1/2 is achieved. Then the error probability is $\frac{2}{r+1}$, which is strictly greater than $\frac{1}{r}$.

Range 2: $[2, r]$ instead of $[1, r]$
Conditional on Case 1, 2, disagreement occurs with probability 0.
Conditional on Case 3, disagreement occurs with probability $\frac{1}{r-1}$.

The best the adversary can do is to go for case 3 with probability $\frac{1}{r-1} > \frac{1}{r}$.

Hence, we can see that $[1, r]$ is better than $[0, r], [2, r]$ as it does not favor some cases over others. The adversary can always go for the least favored case.

Note: when we say that the adversary "goes for" case i, we mean that for e.g. in case 1, the adversary drops all messages from P1 to P2, but allows at least one message from P2 to P1 to pass. At this point in the adversary's perspective, bar is unknown, so in the adversary's perspective, bar still follows a probability distribution uniform over $\{1, \ldots, r\}$. (Something something information interpretation of probability)

Note: Another way to understand this:
We consider a deterministic adversary, i.e. the adversary's choice of communication channel behavior is not a random event. Then we can imagine that the adversary decided prior to the algorithm execution on a particular schedule. For e.g. the adversary decides that the schedule "goes for" Case 1. It is only after this, that the variable bar is chosen. Now it is very clear that for the range $bar \in [1, r]$, the probability of Case 1 disagreement occurring is equal to $\frac{1}{r}$. For the range $bar \in [0, r]$, the probability becomes $\frac{2}{r+1}$.

There is no benefit if the adversary was randomized instead. Because,

- if there is indeed a preferred strategy (e.g. amongst Case 1,2,3), then there is no point in randomizing; the adversary should just keep choosing the preferrred strategy with probability 1
- if there is no preferred strategy, then randomization does not help either. i.e. Taking an average of equal values would yield the same value.

Model 3

- Node crash failure
- Asynchronous timing model

Some pointers on the proof of the FLP theorem.

Some FLP assumptions
Suppose there exists a consensus protocol that can achieve consensus with

- At most 1 node failure
- State machine model, state transitions are due to events of the form (p, m), where m is a message for receiver process p.
    - Global state = Local state for each process + Messages present in message system
    - Note that the message system is not part of the local state, so a process can only "query" the message system for an event (p, m). The message system may very well not deliver anything of substance, i.e., m = null. Essentially, this is a non-blocking receive.
- Unboundedness (but finiteness) of time to deliver a particular event (p, m) in the message system. This models the async timing model.

The state machine model

The state machine model models state transitions in the form of applying a linear sequence of events (p, m) to a single process p. This makes sense as we can always apply a total ordering to events. Furthermore, we can always form a total order that respects the happened-before partial order (even if there are infinitely many events, this can be proven by axiomatic set theory).

The proof

The high level statement of the proof is that: We can form a (adversarial) schedule such that at any point in time,

1. All non-faulty processes can get to increase the number of steps they run. (So that in the long run, as time goes to infinity, the number of steps taken by non-faulty processes goes to infinity)
2. All messages in the message system (especially those intended to non-faulty processes) gets delivered.
3. The global state remains bivalent.

Conditions 1 and 2 ensure that our schedule is valid. Condition 3 ensures that the system is unable to decide, and hence fails termination, i.e. the protocol is incorrect.

This can be proven by induction.

- The use of round robin ensures that each non-faulty process gets to take at least one step.
- The choice on an oldest message for a process ensures all messages get delivered.

Lemmas employed

- Lemma 1 is used as a base case in the induction process.
- Lemma 2 is used to prove Lemma 4
- Lemma 3 is quite trivial and is only used to show that it is legal to talk about Lemma 4.

- Lemma 4 is used in the induction step of the FLP proof.

In lecture, Lemma 4 makes use of 4 claims. In order to obtain a contradiction, we assume that at some bivalent state G, all states G' reachable from G without applying some event e are such that e(G') is univalent. This assumption is used in Claim 4, as we will demonstrate.
Claim 4: Let W be the set of states reachable from G without applying e. Then there exists states F_0, F_1 such that

- $e(F_0)$ is 0-valent
- $e(F_1)$ is 1-valent
- either F_0 is the parent of F_1 or F_1 is the parent of F_0

By claims 1, 2, 3, there exists some state G_0 such that $e(G_0)$ is 0-valent and some state G_1 such that $e(G_1)$ is 1-valent.
We consider the chain of states from G that leads to G_0 and G_1 respectively.

$G_{0,1}, G_{0,2}, \ldots, G_{0,m}$ where $G_{0,1} = G$ and $G_{0,m} = G_0$
$G_{1,1}, G_{1,2}, \ldots, G_{1,n}$ where $G_{1,1} = G$ and $G_{1,m} = G_1$

By our assumption, $\forall 1 \leq i \leq m, e(G_{0,i})$ is univalent. Similarly, $\forall 1 \leq j \leq n, e(G_{1,j})$ is univalent.
WLOG, $e(G) = e(G_{0,1}) = e(G_{1,1})$ is 0-valent. Then considering $e(G_{1,1}), e(G_{1,2}), \ldots, e(G_{1,n})$, since $e(G_{1,1})$ is 0-valent and $e(G_{1,n})$ is 1-valent, we can find some adjacent $e(G_{1,j}), e(G_{1,j+1})$ such that $e(G_{1,j})$ is 0-valent and $e(G_{1,j+1})$ is 1-valent.
Similarly if we had assume $e(G)$ is 1-valent, we simply consider $e(G_{0,1}), e(G_{0,2}), \ldots, e(G_{0,m})$ instead.

Model 4

Note that the timing model is synchronous.
According to Wikipedia, the difference between the Byzantine failure model and the node crash failure model is that

- In the Byzantine failure model, the faulty node may deliver inconsistent information to other nodes/observers, so that its faulty nature may not be detectable.
- In the node crash failure model, the faulty node simply stops broadcasting information at some point in time. In a synchronous timing model, when rounds are used, after a node v crashes, eventually in all future rounds, v will no longer send any info to any other node. The absence of a broadcast from v allows other nodes to detect that v has crashed.

Self-stabilization

A major distinction between a persistent algorithm and a non-persistent algorithm is that the persistent algorithm not only runs all the time, but only makes local changes. For instance, the persistent spanning tree algorithm, upon encountering a fault at some place, will only introduce changes adapting to the fault in that place where the fault occurs. In comparison, a non-persistent spanning tree construction algorithm would involve running from the very beginning and build the tree globally. This may be inefficient.

Persistent spanning tree algorithm

Initial conditions:

- Let V be the set of nodes. Let node $v_0 \in V$ be "special", in that our goal is to maintain a spanning tree rooted at v_0.
- Each node v has two variables, d (distance estimate) and p (parent), which can be initialized to anything, except that $d \geq 0$.
- The graph G is connected and modelled as an undirected graph. Each edge $e = (u, v)$ represents a bidirectional communication channel between u and v. In wireless networking, this may mean that nodes u, v are in proximity of each other.

Algorithm:
v_0 sets: $v_0.d \leftarrow 0, v_0.p \leftarrow \texttt{null}$.
Periodically, all other nodes $v \neq v_0$ queries each neighbor for their value of d. Let u be the neighbor that replies with the smallest d value. Then $v.d \leftarrow u.d + 1, v.p \leftarrow u$.

Proof of correctness. Define a phase as in the textbook, an interval over which all nodes has taken at least one action. Denote $v.L$ as the minimum distance between node v and node v_0.
Proposition Let H be the $\max\{v.L : v \in V\}$. Then forall $1 \leq r \leq H + 1$, after r phases, $\forall v \in V$

- $v.L \leq r - 1 \implies v.d = v.L$
- $v.L \geq r \implies v.d \geq r$
  Denote the rth statement by $P(r)$.

The argument is by induction on r. In the base case, after one phase, v_0 sets its d to 0, hence this meets the first condition. All other nodes will add one to the minimum value they receive, hence their d value must be at least 1. This meets the second condition.
Suppose after r phases, both conditions hold. Consider the $r + 1$th phase, which consists of a certain number of actions by the nodes in V. Note that by definition of a phase, any node must take at least one action. Note that these actions are possibly concurrent. The argument will take the following form:

1. The properties under $P(r)$ are maintained by each action for any node v.
2. When a node v takes an action, $P(r + 1)$ will hold for v.
3. The properties under $P(r + 1)$ are maintained by each action for any node v for which $P(r + 1)$ already holds.

Lemma For a node v with distance L, its neighbors can only have distance from v_0 amongst $\{L - 1, L, L + 1\}$. Additionally, v must have at least one neighbor with distance $L - 1$.

Lemma Suppose $P(r+1)$ holds for a particular node v. Then $P(r)$ also holds for v.
The proof follows by a simple case analysis

- If $v.L \leq r-1$, then we trivially have $v.d = v.L$ by $P(r+1)$
- If $v.L = r$, then $v.d = v.L = r \geq r$
- If $v.L > r$, then $v.d \geq r+1 \geq r$

Hence, $P(r)$ holds for v. $\square$

Steps 1 and 2: Maintenance of $P(r)$. Let $v \in V$ for which $P(r)$ holds. We consider cases, in the event that v has taken an action.

- Suppose $v.L < r$. We consider the neighbors u of v.
  - $u.L = v.L - 1$. Then by $P(r)$, $u.d = u.L = v.L - 1 < r$. (Furthermore, such a u must exist.)
  - $u.L \geq v.L$. Then by $P(r)$, $u.d = u.L \vee u.d \geq r$. In either case, $u.d > v.L - 1$.
  - Hence, the minimal neighbor value must be $v.L - 1$. Hence, $v.d \leftarrow v.L - 1 + 1 = v.L$.
- Suppose $v.L = r$
  - $u.L = r - 1$. Then by $P(r)$, $u.d = r - 1$
  - $u.L \geq r$. Then by $P(r)$, $u.d \geq r$
  - Hence, the minimal neighbor value must be $r - 1$. Hence, $v.d \leftarrow r = v.L$.
- Suppose $v.L > r$
  - It is not hard to see that $v.d \geq r + 1$.
    Hence, if v took an action, then $P(r+1)$ holds for v after the action. Since $P(r+1) \implies P(r)$, $P(r)$ holds for v.
    Otherwise, if v took no action, then $P(r)$ trivially continues to hold as $v.d, v.p$ remains constant.

Remark Notice that we have combined maintenance with going from $P(r)$ to $P(r+1)$ for nodes that took an action.

Step 3: Maintenance of $P(r+1)$. Let $v \in V$ where $P(r+1)$ holds for v. Suppose v took an action. By step 1, all neighbors of v satisfy $P(r)$.

- Suppose $v.L < r + 1$. Consider neighbors u of v. There exists a neighbor with $u.L = v.L - 1 < r$ and so by $P(r)$, $u.d = u.L$. It is not hard to show this is minimal amongst all neighbors. Hence, $v.d \leftarrow u.d + 1 = v.L$.
- Suppose $v.L \geq r + 1$. Then $u.L \geq (r+1) - 1 = r$. Since $P(r)$ holds for u, we have $u.d \geq r$, hence $v.d \leftarrow$ some value $\geq r + 1$.

If v took no action, then $P(r+1)$ trivially continues to hold.

Combining steps 1, 2, 3, we see that given $P(r)$ holds for all $v \in V$, after 1 phase, $P(r+1)$ holds for all $v \in V$. To be precise, we argue that for a particular node $v \in V$:

- One of the actions is taken by v itself, so after this $P(r+1)$ holds for v. (Using step 2)

- After the remaining actions, $P(r+1)$ continues to hold for v. (Using step 3)

Note: step 1 is used by steps 2 and 3, so step 1 is essential.

At termination, we let $r = H+1$, so $\forall v \in V, v.d = v.L$. We thus have a shortest path spanning tree. (To be more rigorous, we also need to argue that the parent pointers are correct.) $\square$

We can optimize this proof further. For a node w, we denote $P(r,w)$ as $P(r)$ holds for w. Note that $P(r+1,w) \implies P(r,w)$. We can prove the following. If $\forall w, P(r,w)$,

- If v takes a step, then $P(r+1,v)$. This is similar to how step 2 is proven in the previous proof.
- If v doesn't take a step, and $P(r,v)$, then $P(r,v)$ continues to hold. This is trivial since $v.d$ doesn't change.
- If v doesn't take a step, and $P(r+1,v)$, then $P(r+1,v)$ continues to hold. This is trivial since $v.d$ doesn't change.

So in some sense, we have compressed steps 2 and 3 into one, which results in a shorter proof. Now, to make use of these 3 statements, we can argue by induction. Suppose $\forall w, P(r,w)$. Note that after any step, we will still have $\forall w, P(r,w)$ since $P(r+1,w) \implies P(r,w)$.
Pick any node v. Across the $r+1$-th phase, v must have take at least one step. Right after that step, by the first statement, $P(r+1,v)$. From then onwards, if a step is taken by v, $P(r+1,v)$ will hold since $P(r+1,v) \implies P(r,v)$ and we apply statement one. If a step is taken by some other node, then the third statement implies $P(r+1,v)$. Hence, eventually, at the end of the $r+1$-th phase, $P(r+1,v)$ will hold. Since v was arbitrary, $\forall w, P(r+1,w)$. $\square$

Proposition The parent pointers of the nodes are correct and result in a shortest path rooted spanning tree.

Proof

- Each node (other than the special node) chooses exactly one node to be its parent. Hence, there are $n-1$ edges in the graph, where n is the number of nodes.
- The parent of a node v has distance $v.L-1$, so by induction, the edges always form a shortest path to the special node.
- Furthermore, this graph is connected. Take any two nodes u, v. u can first move to the special node via the parent pointers, then move to v by taking reverse edges. Hence, if we treat the edges as undirected, then any two nodes has a path between them.

Since we have a connected graph with $n-1$ edges, this must be a tree. Since each node has a path to the special node, this tree is rooted (at the special node). Since the path taken is of the shortest distance, this is a shortest path tree. $\square$