Jin Wei CS4231 Finals Notes

Mutual Exclusion

- (*required) mutual exclusion: no more than one process in the critical section
- progress: if one or more process wants to enter and if no one is in the critical section, then one of them can eventually enter the critical section
  - aka resource is fully utilised
  - only need to consider sections of the algorithm that *can be blocked*
- no starvation: if a process wants to enter, it can eventually always enter

no starvation => progress

Peterson's Algorithm

Mutual exclusion for 2 processes. Can be extended to n processes by using a tournament tree of Peterson's Algorithm: must acquire from leaf to root and release from root to leaf!

```
bool wantCS[0] = false;
bool wantCS[1] = false;
int turn = 0;

// Process 0
void RequestCS() {
    wantCS[0] = true;
    turn = 1; // let it be `other`'s turn to prevent starvation
    // wait if `other` wants and it's their turn
    while (wantCS[1] == true && turn == 1);
}
void ReleaseCS() {
    wantCS[0] = false; // stop wanting it
}

// Process 1 is mirrored
```

PETERSON'S ALGORITHM CORRECTNESS

Mutual Exclusion

Proof by contradiction

- Case turn == 0 when both are in CS
  - then `wantCS[0] = wantCS[1] = true`
  - since `turn == 0`, P0 executed `turn = 1` before P1 executed `turn = 0`
  - which means P1 must have seen `wantCS[0] == false` because `turn == 0` currently
  - but `wantCS[0] = true` was set by P0
- Case turn == 1 is symmetric

Progress

- if both want to enter
  - Case turn == 0
    - then P0 can enter
  - Case turn == 1 is symmetric
- if only one wants to enter, WLOG P0
  - `wantCS[1] = false`, P0 will fall through

No starvation

- if P0 is waiting and P1 in CS
  - after P1 exits the CS, it will set `wantCS[1] = false` so P0 can fall through
  - if P1 immediately wants to re-enter and sets `wantCS[1] = false` immediately, it is still fine because it must also set `turn = 0` before falling through itself (and it can't fall through since `turn = 0`)

Lamport's Bakery Algorithm

Mutual exclusion for n processes.

for n processes:

- get a number
- get served when all processes with a lower number has been served

```
bool choosing[N]; // choosing[i] = true <=> process i is getting a number
int number[N]; // number[i] => number of process i (if number is 0, process doesn't wanna be served)

void RequestCS(int id) {
    choosing[id] = true; // "weak mutex"
    for (int j = 0; j < n; j++) {
        temp = number[j];
        if (temp > number[id]) {
            number[id] = temp;
        }
    }
    number[id]++; // max(everyone else) + 1
    // won't be correct because everyone is racing
    // but it's fine

    // wait for processes with a smaller number OR *might* have a smaller number
    for (int j = 0; j < n; j++) {
        while (choosing[j] == true);
        // compare by (number, id)
        while (number[j] != 0 && Smaller(number[j], j, number[id], id));
    }
}
void ReleaseCS(int id) {
    number[id] = 0;
}
```

Dekker's Algorithm

```
bool wantCS[0] = false;
bool wantCS[1] = false;
int turn = 1;

void RequestCS(int i) {
    // turn doesn't change in RequestCS

    int j = 1 - i;
    wantCS[i] = true;
    while (wantCS[j]) {
        if (turn == j) {
            // temporarily release so the other guy can enter
            wantCS[i] = false;
            while (turn == j);
            wantCS[i] = true;
        }
    }
}

void ReleaseCS(int i) {
    turn = 1 - i;
    wantCS[i] = false;
}
```

Synchronisation Primitives

Semaphore Semantics

Semaphores can be used to implement monitors, and vice versa.

```
// both of these operations are done ATOMICALLY!!
void P() {
```

```
    if (value == false) {
        add myself to queue;
        block;
    }
    value = false;
}
void V() {
    value = true;
    if (queue is not empty) {
        // exactly **ONE** process is woken
        wake up one *arbitrary process* on the queue;
    }
}

void RequestCS() { P(); }
void ReleaseCS() { V(); }
```

The queue is **not** FIFO, it has an **arbitrary** ordering. *unless otherwise specified by the API.

Dining Philosopher Problem

Need to avoid cycles or have a total ordering of the chopsticks to **prevent deadlocks**

Monitor Semantics

Semaphores can be used to implement monitors, and vice versa.

Every object in Java is a monitor.

```
synchronized (object) { // enters monitor
    object.wait();
    object.notify();
    object.notifyAll();
} // exits monitor
```

- enter monitor
  - if no one is in the monitor: I will enter
  - otherwise: I enter the `monitor-queue` and block
- exit monitor
  - if `monitor-queue` is non-empty: unblock one arbitrary process
- `object.wait()` -> this is one atomic operation
  - add to `wait-queue`
  - and block
- `object.notify()`
  - if `wait-queue` is empty, pick one arbitrary process from `wait-queue` and unblock it
- `object.notifyAll()`
  - unblock **all** processes on the `wait-queue`
  - this is only for Java-style, not Hoare-style

assume that the `monitor-queue` is starvation-free for CS4231 because we have no control over this.

> NOTE: the `wait-queue` is not FIFO, so, we must maintain our own FIFO queue if we need FIFO

- Hoare-style: when you `notify`, someone else will takeover
  - possibly can use `if (x == 1) object.wait();`
- Java-style: when you `notify`, you will continue running
  - the other guy wakes up and immediately re-enters the queue (nothing will be done!!)
  - should probably use `while (x == 1) object.wait();`

Nested Monitors

Nested monitors in Java are nasty (other implementations might differ). See the following.

```
// P0
synchronized (x) {
    synchronized (y) {
        y.wait();
```

```
        // only monitor-lock on y is released
        // still holds monitor-lock on x
    }
}

// P1
synchronized (x) { // P1 cannot acquire monitor-lock on x
    synchronized (y) {
        // never runs
        y.notify();
    }
}
```

Use flags to avoid having to use nested monitors. See the starvation-free Reader-Writer solution below.

Producer-Consumer Problem

```
void producer() {
    synchronized (buffer) {
        if (buffer.isFull()) {
            buffer.wait();
        }
        add to buffer;
        if (buffer WAS empty) {
            buffer.notify();
        }
    }
}

void consumer() {
    synchronized (buffer) {
        if (buffer.isEmpty()) {
            buffer.wait();
        }
        remove from buffer;
        if (buffer WAS full) {
            buffer.notify();
        }
    }
}
```

Note that notification will be lost if *no one* is waiting! Doesn't matter for this Producer-Consumer solution tho.

Reader-Writer Problem

```
// this solution will starve writers!

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

void readFile() {
    synchronized (object) {
        while (numWriter > 0) {
            object.wait();
        }
        numReader++;
    } // must leave the monitor, so other readers can enter!
    write to file;
```

```
    synchronized (object) {
        numReader--;
        object.notify(); // wake up a writer
        // can be proven that only writers are waiting!
        // proof by contradiction: suppose a reader gets notified, then it is blocked, but it shouldn't be
blocked because `numWriter == 0`
    }
}
```

To be starvation-free: maintain an explicit queue and let each thread wait on its own monitor. Avoid waiting on a common object (because there will be no guarantee of FIFO). Use a flag to handle the case where you `notify` before the other guy calls `wait`.

```
// this solution is starvation-free!!

Queue queue;

void writeFile() {
    Writer w = new Writer();

    synchronized (queue) {
        if (numReader > 0 || numWriter > 0) {
            w.okToGo = false;
            queue.add(w);
        } else { // no writers and no readers
            w.okToGo = true;
            numWriter = 1;
        }
    }
    synchronized (w) {
        if (!w.isOkToGo) {
            w.wait();
        }
    }

    write to file;

    synchronized (queue) {
        numWriter = 0;
        if (!queue.isEmpty()) {
            // remove a single writer OR a batch of readers from the queue
            // in a FIFO way
            for (auto request : objects) {
                numWriter++ OR numReader++;
                synchronized (request) {
                    // important because this can be called **between** line 16 and line 17
                    request.okToGo = true;
                    request.notify();
                }
            }
        }
    }
}

void readFile() {
    Reader r = new Reader();

    synchronized (queue) {
        if (numWriter > 0 || !queue.isEmpty()) {
            // queue is only non-empty if there is at least
            // one writer waiting
            r.okToGo = false;
            queue.add(r);
        } else { // no writers waiting or running
            r.okToGo = true;
            numReader++;
        }
    }
    synchronized (r) {
        if (!r.isOkToGo) {
            r.wait();
```

```
        }
    }

    write to file;

    synchronized (queue) {
        numReader--;
        if (numReader > 0) return; // only last reader runs the following code

        if (!queue.isEmpty()) {
            // remove a single writer OR a batch of readers from the queue
            // in a FIFO way
            for (auto request : objects) {
                numWriter++ OR numReader++;
                synchronized (request) {
                    // important because this can be called **between** line 16 and line 17
                    request.okToGo = true;
                    request.notify();
                }
            }
        }
    }
}
```

Consistency

Consistency specifies what behaviour is allowed when a shared object is accessed by multiple processes.

Something is consistent if it satisfies the specification.

Sequential Consistency

Results should be the same as if all operations are executed in some sequential order (as if it was ran on a simple single-core system).

Operation: a single invocation-response pair of a single method of a single shared object by a process.

- Two invocation events are the same if invoker, invokee, parameters are the same.
- Two response events are the same if invoker, invokee, response are the same.

A **history** H is a sequence of invocations and responses ordered by wall clock time. All invocations in H must have their responses in H too.

A history H is **sequential** if

- any invocation is *immediately* followed by its response
  - no interleaving
- otherwise: it is *concurrent*

A history H is **legal** if

- all responses satisfy the sequential semantics (acts like there's one process)

Sequential $\subseteq$ Legal.

- (H | p) is process p's subhistory of H.
  - this is always sequential (because of process order)
- (H | o) is object o's subhistory of H.

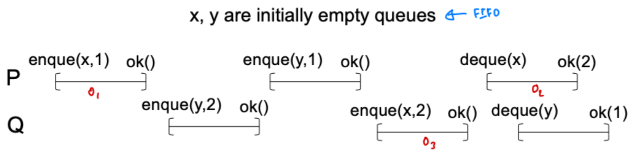
Two histories are **equivalent** if they have *exactly the same set of events*

- all responses are the same (because responses are part of the event)
- ordering of events can differ

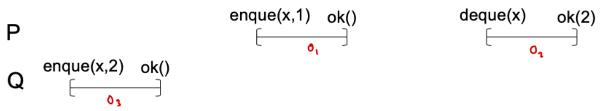
A history H is **sequentially consistent** if it is *equivalent* to (i.e. same result as) some *legal sequential history* S that preserves *process order* (partial ordering among all events).

- same result as some single-core version (invocation-response pair happens immediately)
- preserve program order

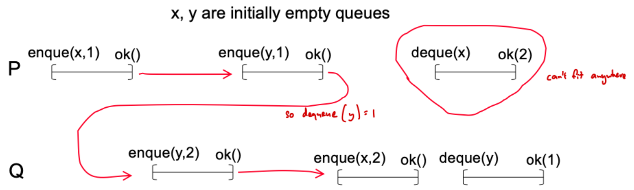
Sequential consistency is NOT a *local property*.



H | x is sequentially consistent:



Similarly, H | y is sequentially consistent as well



But H is not sequentially consistent

Linearizability

A history H is **linearizabile** if

1. execution is *equivalent* to (i.e. same result as) some execution such that each operation happens *instantaneously* (called linearization point).
2. execution is *equivalent* to (i.e. same result as) some *legal sequential history* S AND S preserves the external order in H
   - aka sequentially consistent + preserves external order
   - external order (occurred-before order): `o1 < o2` if response of o1 happens before invocation of o2
     - aka we can *only reorder things that are concurrent*

(^ two equivalent definitions)

Sequential Consistency $\subseteq$ Linearizability

Linearizability is a *local property*: H is linearizable <=> for any object x, (H | x) is linearizable. Proof:

- $\Longrightarrow$
  - construct a graph, then it is easy to show
- $\Longleftarrow$
  - construct graphs for all (H | x)
  - show that we can join them with the cross edges
    - Lemma: resulting directed graph is acyclic (prove using contradiction)
  - any topological sorting gives us a linearizable H

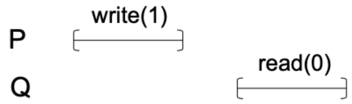
Consistency for Registers

A register is **atomic** if the implementation always ensures *linearizability* of the history.

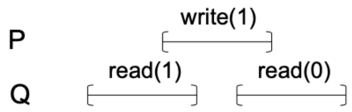
A register is called **regular** if

- when *a read does not overlap with any write*, the read returns the value written by *one of the most recent writes*
- when *a read overlaps with one or more writes*, the read returns the value written by *one of the most recent writes* OR the value written by *one of the overlapping writes*

Regular DOES NOT imply Sequential Consistency.
Sequential Consistency DOES NOT imply Regular.



Sequentially consistent
but not regular

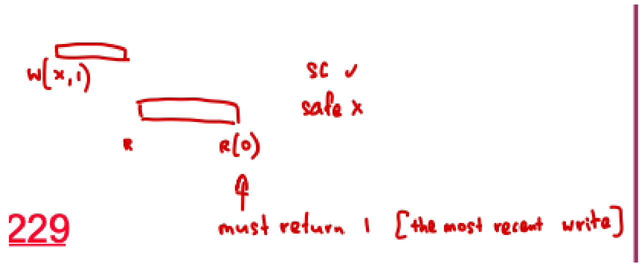


Regular but not
sequentially consistent

A register is **safe** if

- when *a read does not overlap with any write*, the read returns the value written by *one of the most recent writes*
- when a read overlaps with one or more writes, it can return **anything**

Safe DOES NOT imply Sequential Consistency.
Sequential Consistency DOES NOT imply Safe.



To find "most recent writes":

1. find the write with the latest response time
2. any write that is concurrent with (1) is "most recent"

Clocks

Assumes the following:

- processes can do these:
  - **local computation**
  - **send** / **receive** a *single message* to a *single process*
    - no atomic broadcast (must be emulated using point-to-point messages)
- communication model
  - point-to-point messages
  - error free (no corruption + no message loss) + infinite buffer
  - potentially out of order

Goal of clocks: Capture event ordering even if the users do not have physical clocks.

Happened-before

Happened-before relationship (a partial order)

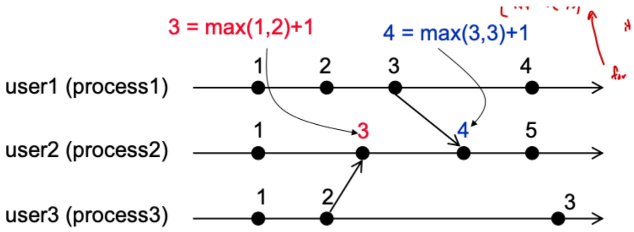
- process order
- send-receive order
- transitivity

Note that *concurrent with* is **not** transitive!

(Integer) Logical Clock

Each process maintains a single **integer**.

- increment local_time at each *local computation* and *send event*
- when sending a message, attach local_time
- when receiving a message, local_time = max(local_time, sender.time) + 1
  - any `f` works as long as `f(x, y) > x` and `f(x, y) > y`



Event s happen before $t \implies$ logical clock of $s <$ logical clock of t

^ Use the definition to prove

logical clock of $s <$ logical clock of $t \implies s$ happened before t OR they are *concurrent*

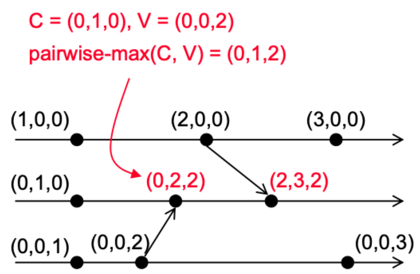
Vector Clock

Each process maintains a **vector** of size n, where n is the number of processes.

`local_vector[i]` is known as the *principle entry*.

- increment `local_vector[i]` at each *local computation* and *send event*
- when sending a message, attach local_vector
- when receiving a message:
  - local_vector = pairwise_max(local_vector, sender.time)
  - `local_vector[i]++`

c[i] : principle entry



Event s happen before $t \iff$ vector clock of $s <$ vector clock of t

^ ($\implies$) Use the definition to prove
^ ($\impliedby$) Consider two cases: (1) on same process; (2) there was a sequence of events to propagate the clock

- vectors $v1 < v2 \implies$ all fields in $v1$ are $\leq$ all fields in $v2$ **AND** at least one field in $v1$ is $<$ than the corresponding field in $v2$
- this $<$ relationship is not a total order. Example: $(1,0)$ and $(0,1)$
- this $<$ relationship is transitive (because happens-before is transitive)

To convert Vector Clock to Logical Clock: it is sufficient to take the **summation** of all the entries in the vector clock. (taking maximum is not correct)

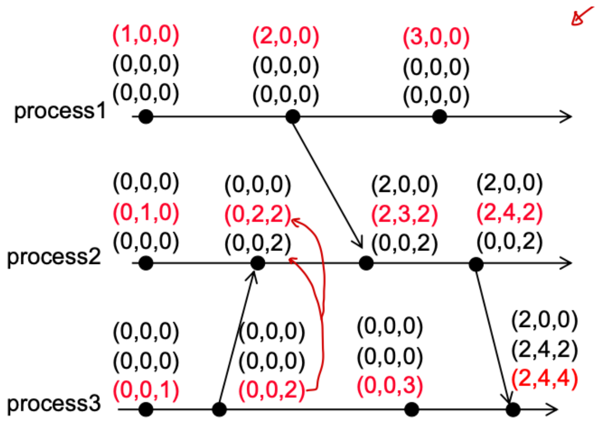
Matrix Clock

You are able to know what *other processes* (definitely) have seen.

Each process maintains n vector clocks: one for each process.

`local_matrix[i]` is the *principle vector*.

- for the principle vector, we do exactly the same thing as in Vector Clock
- for other vectors in the matrix:
  - perform pairwise-max to update local copy



Total Ordering Mechanism

Use the partial order created by one of the clock systems. Then, tie-break with one of the following ways:

1. Tie-break using the process ID.
    1. unfair because it would favour those with a lower process ID
2. Tie-break randomly.
    1. Tie-break by using the time to seed a random number generator.
        1. You have to re-seed it every time, or you will run into the same problem as solution (1).
    2. Generate a random ordering of the process IDs.
    3. Use this random ordering to tie-break.

Snapshot

Global Snapshot: a set of events such that if $e2$ is in the set and $e1$ is before $e2$ in *process order*, then $e1$ must be in the set (intuitively: a snapshot of local states on n processes such that the global snapshot *could have happened* sometime in the past)

Consistent Global Snapshot: Global Snapshot such that if $e2$ is in the set and $e1$ is before $e2$ in *send-receive order*, then $e1$ must be in the set
(alternatively: a set of events such that if $e2$ is in the set and $e1$ *happened before* $e2$, then $e1$ must be in the set)

For any *prefix of a process*, it is always possible to construct a Consistent Global Snapshot:

- such that *all* events in that prefix is in the CGS
- and all events that are *not* in the prefix are not in the CGS.

If G and H are both CGS, $G \cap H$ and $G \cup H$ are both CGS.
Proof by playing with definition of set intersection, union + CGS.

Chandy & Lamport's Snapshot Protocol

NOTE: messages are ordered and guaranteed to be FIFO through message numbers!

Each process is either

- red (has taken local snapshot)
- OR white (has not taken local snapshot)

Trigger the protocol on one process p:

1. j turns from white -> red
2. it immediately sends out $n-1$ *Marker* messages to all other processes
3. upon receiving first *Marker* message, a process will turn from white -> red and propagate the *Marker* message

Need to capture "on-the-fly" application messages.

There are only four possible cases of a message M in relation to capturing snapshot

1. case 1 (M is captured by both)
    - sent before local snapshot on sender
    - received before local snapshot on receiver
2. case 2 (M is not captured by both)

    - sent after local snapshot on sender
    - received after local snapshot on receiver
3. case 3 (not CGS)
    - sent after local snapshot on sender
    - received before local snapshot on receiver
        - impossible because of FIFO
4. case 4 (need to handle this case separately *)
    - sent before local snapshot on sender
    - received after local snapshot on receiver
        - possible because receiver turned red from someone else before receiving this sender's marker

Case 4 is handled by appending these on-the-fly messages to the local snapshot.

Lamport's Logical Clock to compute CGS

- cut(e) = { f | f has smaller logical clock value than e }
- for any event e, cut(e) is a consistent global snapshot
    - just use definition of GS and CGS
        - because of process order & send-receive order
        - this will be maintained by the logical clock algorithm

---

Ordering

Causal Order

Causal order: if $s1$ happened before $s2$, and $r1$ and $r2$ are on the same process, then $r1$ must be before $r2$.
(pessimistically assume that $s1$ caused $s2$)

Each process maintains a n by n matrix M (NOT the matrix clock).

$M[i, j]$ is the number of messages sent from i to j as known by the local process

- if i sends a message to j:
    - on i: $M[i, j]++$
    - piggyback M
- upon receiving the message with T, set M = pairwise-max(M,T) **if**
    - $T[k, j] \leq M[k, j]$ for all $k \neq i$
    - AND $T[i, j] = M[i, j] + 1$
- Intuitively, $M[i, j]$ must be sent sequentially and I have seen whatever stuff that I'm supposed to have seen

Proof:

- show that if s1 happens before s2, r2 will not happen before r1 (show using properties of matrix)
    - case 1: s1 and s2 on same process
    - case 2: s1 and s2 on different processes
- show that at any given time (when all messages have been received, but not delivered), at least one message can be delivered, induction will handle everything else
    - define set of successor messages:
        - the next-to-deliver message from each sender (if the sender has an undelivered message)
    - define top successor message:
        - at least one of the successor messages has no other send events that happened-before it
    - any of these top successor messages can be sent

Causal Ordering of Broadcast Messages

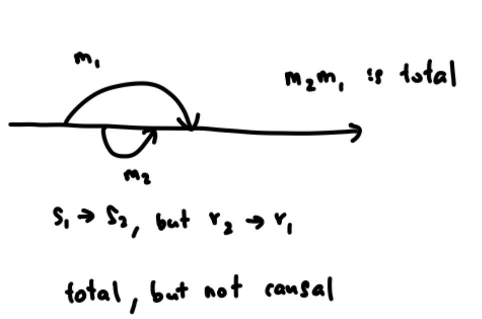
Exactly the same as point-to-point.

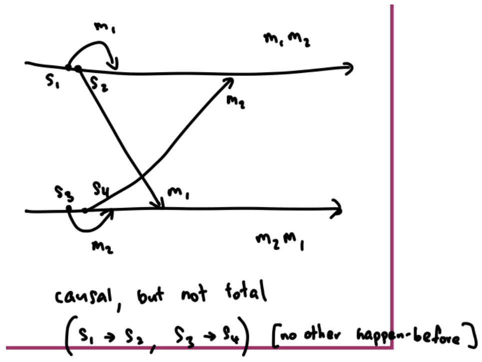
Total Ordering of Broadcast Messages

Total ordering only applies to broadcast messages.

Atomic broadcast: all messages delivered to all processes in exactly the same order. (impossible in asynchronous system)

Total does NOT imply Causal
Causal does NOT imply Total





Coordinator for Total Ordering

- send message to coordinator
- coordinator assigns sequence number
- coordinator broadcasts to all
- messages delivered according to sequence number

BAD! because this is centralised

Such a Total Order would not necessarily be Causal Order because messages sent to the coordinator are not necessarily FIFO.

Skeen's Algorithm for Total Order Broadcast

Every process maintains a logical clock + message buffer for undelivered messages.

Messages are delivered from the buffer if:

- all messages have been numbered
- the message has the smallest number

For me to send a message:

1. Broadcast to all processes
2. All processes will reply with their logical clock value
3. I pick the maximum as the message number
4. I broadcast this back

Leader Election

Leader Election on Anonymous Ring

- Ring topology
  - simplest topology is line topology
  - but ring topology doesn't break from a single edge failure
  - so, we consider ring topology
- No unique identifiers
  - the nodes are completely indistinguishable

Leader Election on anonymous ring is **impossible** with deterministic algorithms.

- same initial state
- same algorithm on each node
- same steps are taken
  - consider worst case of all nodes executing the same step at the same time and speed
  - we only need to consider the algorithm failing for one case
- same final state
  - either the algorithm fails because no leader is found
  - or everyone claims they're the leader.

Leader Election on an Anonymous Ring with Unknown Size

This is impossible, even with randomised algorithms.

- assume that such a algorithm exists

- model the randomised algorithm as a deterministic algorithm that takes in a random bit string as one of its input
- consider two rings
  - ring 1: exactly one node P1
  - ring 2: two nodes P1 and P2
- terminates with probability 1 => exists some input that P1 can terminate (for ring 1)
- we use the same input for ring 2: (note: both rings are indistinguishable)
  - then, P1 and P2 must both declare they are the leaders
  - contradiction

Leader Election on an Anonymous Ring with Known Size

- done in phases, all nodes start at phase 1
  - each message has the phase number attached
- in each phase:
  - each node picks a random ID, then run Chang-Roberts
  - winners proceed to the next phase; losers will only forward messages from now on
    - winner <=> sees its own ID after exactly n hops
  - stop when there is a single winner

Protocol will terminate with a probability of 1 (doesn't mean it *always* terminates). Prove using good phases and probability. After i -th good phase, there are $n - i$ winners left.

Leader Election on a Ring (Chang-Roberts Algorithm)

Each node has a **unique identifier**.

Nodes only send messages clockwise.

Each node acts independently on their own.

- a node sends election message with its own ID clockwise
- election message is forwarded if message.ID > own.ID
  - else ignore it
- a node becomes the leader if it sees its own election message

Performance:

For distributed systems, network communication is the bottleneck! Performance is described by message complexity: total number of messages sent by all the nodes.

- in the best case, with n rings: $n + n - 1 = \Theta(n)$ messages
  - consider sorted clockwise
  - n messages sent / forwarded (of the winner's election message)
  - $n - 1$ messages, one each by the other nodes
- in the worst case, with n rings: $(n * (n + 1))/2 = \Theta(n^2)$ messages
  - consider sorted anti-clockwise
- in the average case, $O(n \log n)$ (probability ***MATH***)

Leader Election on General Graph (n is known)

- Complete Graph
  - send own ID to all other nodes
  - wait for all $n - 1$ other IDs
  - if you're the biggest, you win
- Any connected graph
  - flood your ID to all other nodes
    - with forwarding of message
    - won't relay if relayed before
  - wait for all $n - 1$ other IDs
  - if you're the biggest, you win

Leader Election on General Graph (n unknown)

- Complete Graph
  - no such case because you're connected

- Any connected graph
  - calculate the number of nodes (request-response spanning tree construction)
    - construct a spanning tree, rooted at whoever wants to count the nodes
    - goal: each node will know its parent and children (such that it's a tree!!)
      - node X will "child request" all its neighbours
      - the neighbour will return "YES" or "NO"
        - a node can only say "YES" to one parent
        - a node must say "NO" to all other requests
    - then we just do the good old BFS / DFS on the spanning tree
      - don't send messages to non-tree edges (if the node replied "NO")
  - resolve to previous case of known n

Distributed Consensus

- each node has an input
- want to agree on a result
- but: nodes can crash, network links can fail (network failure)
  - forever waiting for N-1 results
  - OR node X only sends its results to *some* of the other nodes, resulting in different conclusions

5 versions of Distributed Consensus

1. No node or link failures
   - trivial using all-to-all broadcast
2. Node crash failures; channels that are reliable; synchronous
   - use $(f+1)$-round broadcasting protocol to tolerate f failures
3. No node failures; channels that drop messages (coordinated attack problem)
   - impossible without error
   - use randomised algorithm with 1/r error probability
4. Node crash failures; channels that are reliable; asynchronous
   - impossible, provable using FLP theorem
5. Node byzantine failures; channels that are reliable; synchronous (Byzantine General problem)
   - if $n \leq 3f$, impossible
   - if $n \geq 4f+1$, $(2f+2)$-round protocol
   - in between is solvable using a more complicated protocol (not covered in CS4231)

Distributed Consensus Concepts

Crash Failures

- either running correctly
- or suddenly stop executing anything from then on (no recovery mechanism)

Byzantine Failure

- the node goes rogue, can do anything
  - real world: malicious actors or hardware failure + passes CRC

Reliable Channels

- no messages are dropped
- all messages are eventually sent

Unreliable Channels / Link Failure

- channels can drop any arbitrary unbounded number of messages

Synchronous Timing Model

known upper bound on message delay and node processing delay (aka **accurate failure detection**)

- allows for inter-locked rounds (lockstep rounds) -> enables clear progress
  - every process does some local computation
  - every process sends one message to every other process

        - can be empty to "do nothing"
    - every process receives one message from every other process

Implementing Rounds with Synchronous Model

Further assume each process has a physical clock with some bounded clock error.

Set the round duration = message_delay + node_processing_delay + physical_clock_error

> A message sent in a round must be received by the end of the round.

Each message has a round number attached.

Asynchronous Timing Model

Process delay and message delay are finite but unbounded (no upper bound guarantee).

Thus, impossible to define rounds like synchronous model. No way to tell if the message has *failed or just delayed*.

In real world, it is always possible to define a synchronous timing model, but you might have to set it to a very large value like 1 minute. Latency would be ridiculous, so asynchronous timing models are used instead.

Version 1: No failures

Trivial. Just broadcast to every other node, take the majority of the results (with tie breaking).

Version 2: Node Crash; Reliable Channels; Synchronous

- **Termination**: all nodes (that have not failed) eventually decide
    - proof: obvious since no waiting (well-defined rounds)
- **Agreement**: all nodes that decide should decide on the same value
    - including the nodes that would crash after deciding (because its decision can be used)
    - proof: below
- **Validity**: if all nodes have the same initial input: that value is the only possible decision
    - to avoid a trivial algorithm
    - proof: obvious since `len(S) == 1`

To tolerate f failures: requires $f+1$ rounds of broadcasting!

> Theorem: With f crash failures, any consensus protocol will take $\Omega(f)$ rounds.
>
> - Any deterministic consensus protocol will take at least $f+1$ rounds. (proof is beyond CS4231)

```
Consensus(my_input) {
    S = {my_input}

    // f+1 rounds
    for int i = 1; i <= f+1; i++ {
        send S to all other nodes
        receive n-1 sets
        for each received set T {
            S = S union T
        }
    }

    decide on min(S)
    return decision
}
```

Agreement Proof

- a node is *non-faulty* during round r if it has not crashed by the end of round r
- a round is *good* if there is no node failure during the round

With $f+1$ rounds and f failures, there is at least one good round.

- after a good round, all non-faulty nodes during this round has the same S
    - because everything is broadcasted to everything else
- after this good round, S on these (currently) non-faulty nodes will never change

- because they have received all the information
- therefore, all non-faulty nodes at round $f+1$ has the same S, thus same decision

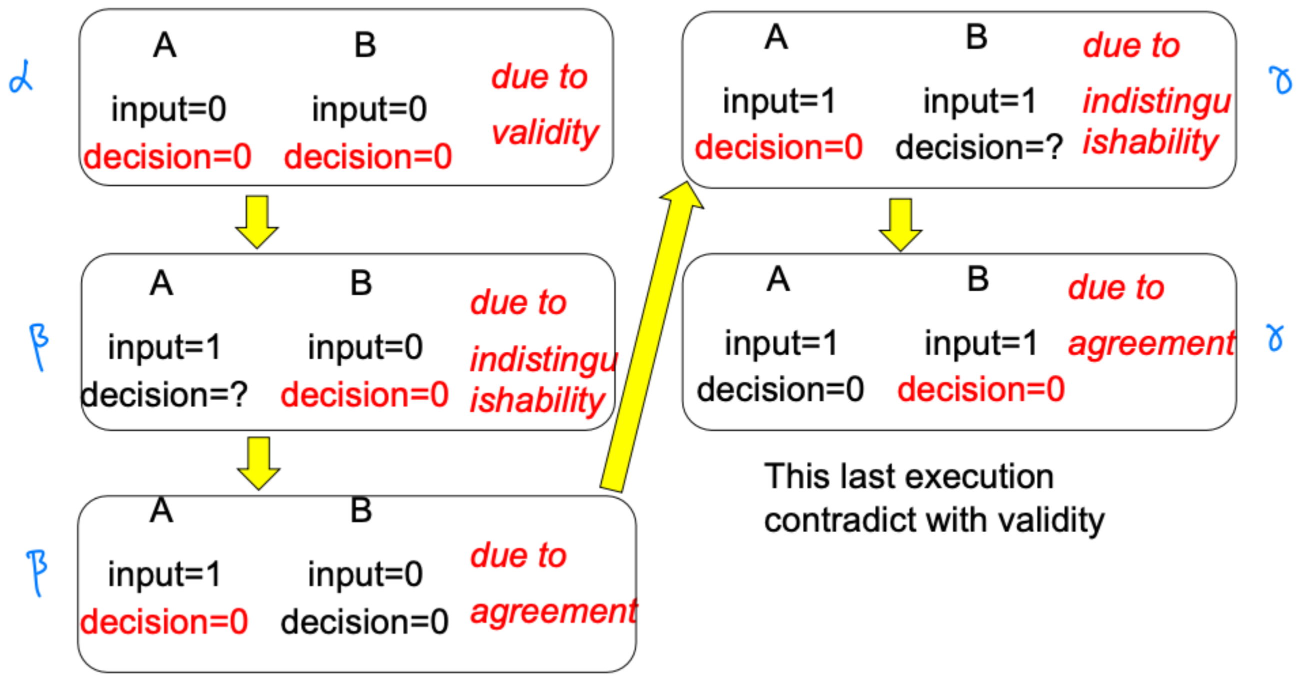
Version 3: No Node Crash; Unreliable Channels

(aka Coordinated Attack Problem)

Impossible to achieve the goals with a *deterministic* algorithm.

Proof:

- For a deterministic algorithm: two executions are *indistinguishable* if the nodes see the same things (messages and inputs)
- Consider the case where all messages are dropped, if the input is the same, the executions are indistinguishable
- Consider this:
  - A (input = 0, decision = 0), B (input = 0, decision = 0) by validity
  - A (input = 1, decision = ?), B (input = 0, decision = 0) by indistinguishability, B must decide on 0 if its input is 0
  - A (input = 1, decision = 0), B (input = 0, decision = 0) by agreement, A must also decide on 0 since B decides on 0
  - A (input = 1, decision = 0), B (input = 1, decision = ?) by indistinguishability, A must decide on 0 if its input is 1
  - A (input = 1, decision = 0), B (input = 1, decision = 0) by agreement, B must also decide on 0 since A decides on 0
  - however, this contradicts with validity



Goals:

- **Termination**: all nodes (that have not failed) eventually decide
  - proof: obvious since no waiting (well-defined rounds)
- **Weakened Agreement**: all nodes that decide should decide on the same value with probability of (1 - error_prob)
  - including the nodes that would crash after deciding (because its decision can be used)
  - proof: below
- **Weakened Validity**: (assume input of 0 or 1)
  - if all nodes start with 0: decision should be 0
  - if all nodes start with 1 and no message is lost throughout the execution, decision should be 1
  - otherwise, nodes can decide on anything
  - weakened validity is not sufficient by itself, consider: (requires weakened agreement)
    - A (input = 1, decision = 1), B (input = 1, decision = 1) by weakened validity with no messages dropped
    - drop the last message (defined using real clock time)
      - sender won't know it's dropped, so it must still decide on 1 by indistinguishable
      - receiver must also decide on 1 by agreement
    - inductively keep dropping the messages until all messages are dropped
    - A and B start with 1 and all messages are dropped, and both decide on 1
    - use the previous argument (with un-weakened validity), but flip 0 and 1

        - then, A and B start with 0 and all messages are dropped, and both decide on 1
            - this is not valid by weakened validity

The Randomised Algorithm

Assume two nodes P1 and P2 (can be generalised) and predetermined number of rounds r.

- P1 picks a random integer `bar` from 1 to r (inclusive)
    - hidden from adversary
- P1 and P2 maintains an integer level `L1` and `L2` respectively
    - `L1` and `L2` can be influenced by the adversary (depending on messages lost)
    - but `L1` and `L2` differ by at most 1
        - proof by cases
- When a message is sent: bar, input and current level is attached.
- When receiving a message:
    - level = other_level + 1

Decision Rule

- at the end of r rounds, P1 decides on 1 iff
    - P1 knows P1 and P2 inputs are both 1
    - *and* $L1 \geq bar$
- at the end of r rounds, P2 decides on 1 iff
    - P2 knows P1 and P2 inputs are both 1
    - *and* P2 knows bar
    - *and* $L2 \geq bar$

Non-agreement if $L1 < bar < L2$, conditions:

- both inputs are 1
- case 1: P1 decides on 1
    - L1 = 1, L2 = 0 because P2 never sees any message (fails only if bar == 1)
    - probability of 1/r
- case 2: P2 decide on 1
    - L1 = 0 because P1 never sees any message, L2 = 1 (fails only if bar == 1)
    - probability of 1/r
- case 3: they see each other, but one decides 0
    - fails only if max(L1, L2) == bar (but L1 != L2, aka one of them is < r)
    - probability of 1/r

In all 3 cases, probability of 1/r that it fails

Version 4: Node Crash; Reliable Channels; Asynchronous

Impossible to solve. Provable using the FLP theorem.

FLP Theorem

FLP = Fischer, Lynch, Paterson (1985)

(in practice, this worst-case scenario would rarely happen and might unblock itself after a while; 2PC and 3PC in DBMS is widely used, but obviously can't handle this)

> In essence, no protocol can accurately detect node failure.

The protocol must still satisfy Agreement, Validity and Termination as previously defined.

Parameters

- Goal: works for any possible deterministic protocol
- each process has local state and two special variables
    - input is `0 or 1`; decision is `null or 0 or 1`
    - decision is initially `null`, and can only change once
- messages in the communication channel can be "on-the-fly" (because async)
    - $\{(p, m) \mid \text{message m is on the fly to process p}\}$
    - all messages are distinct (easy to add on a message-number)
- Send operation = add (destination, content) to message system

- Receive (invoked by process p)
  - Remove (p, content) from the system and return content
  - *or* return null (aka no message)
    - if a message exists, it *must* be returned after a *finite* number of receives (because the channel is reliable: unbounded but *finite* delay)

Does not assume out-of-order or FIFO channels (can implement FIFO using out of order).
Does not assume non-blocking or blocking receives (can implement blocking using non-blocking).

Assumes only *one* crash failure! (sufficient to break any protocol)

Global State

- Global state: includes all process states (even internal states) + message system
- A **step** in the protocol changes one global state to another
  - executes the following one *one* process p
    - receives a message m (can be null)
    - based on p's local state and m, send an arbitrary but finite number of messages
    - based on p's local state and m, change p's local state to some new state

> Note: each step **must** start with receiving a (possibly null) message. Important because we can serialise events across processes this way.

Therefore, each step can be fully described by an **event** (p, m) and the global state G.

An event e **can be applied** to global state G if m is null or (p, m) is in the system.

Classifications of global state G:

- **0-valent** if 0 is the only possible decision reachable from G
  - may not have decided on 0, but will eventually decide on 0
- **1-valent** if 1 is the only possible decision reachable from G
- **univalent** if either 0-valent or 1-valent
- **bivalent** if not univalent

Execution

The **execution** of any deterministic consensus protocol can be abstracted to an *infinite* sequence of events. (infinite because process crash $\iff$ finite steps; just do no-op receives after consensus is achieved)

- FLP theorem proves that there is always some execution that will lead to the protocol not terminating.

A schedule σ is a sequence of events that capture some execution.

- σ **can be applied** to global state G if the events can be applied in order
- $G' = \sigma(G)$ means apply σ to G to get G'
  - σ must be able to be applied to G
- $G2$ is reachable from $G1$ if there is some schedule σ such that $G2 = \sigma(G1)$

Actual Proof

Technique:

- the adversary (we) is the scheduler can
  - pick which messages to deliver
  - which processes will take the next step
- don't need to crash any process
  - but we take advantage of the fact that the protocol must still guard against crash failures
- goal: prevent the protocol from deciding by proving that we can keep it in a bivalent state

Lemma 1

> Lemma 1: For any protocol A, there exists a *bivalent initial state*.

For n processes: there exists $n + 1$ initial states (ignoring ordering of processes): $(0, 0, \ldots, 0)$, $(1, 0, \ldots, 0)$, ... $(1, 1, \ldots, 1)$. (all 0; one 1, then all 0; all 1)

- Assuming no bivalent state exists, there must be two adjacent states S0 and S1 such that S0 is 0-valent, and S1 is 1-valent.
- Let process p be the process that differs between S0 and S1. Consider an execution starting from S0 where process p fails from the start. S0 and S1 are now indistinguishable, so they cannot be "different valent" (i.e. at least one must be bivalent).

Thus, contradiction.

Lemma 2

> Lemma 2: Let $\sigma 1$ and $\sigma 2$ be two schedules such that the set of processes are *disjoint*. Then, for any G that both σ can be applied, $\sigma 1(\sigma 2(G)) = \sigma 2(\sigma 1(G))$.

Proof by induction on $k = max(|\sigma 1|, |\sigma 2|)$; verify both are well-defined

Key idea: start from LHS, inductively swap events until RHS is achieved. Be very careful in the cases.

Lemma 3

> Lemma 3: Let G be a global state, and $e = (p, m)$ can be applied to G. Let W be the set of global states reachable from G without applying e, then e can be applied to any state in W.

aka e can be delayed, but still be applied

Consider:

- m is null: trivial (always can apply null messages)
- m is not null:
  - m is on the fly in G
  - thus, it must still be on the fly in any global state in W

Lemma 4

> Lemma 4: Let G be a bivalent state, and $e = (p, m)$ is any event that can be applied to G. Let W be the set of global states reachable from G without applying e, and $V = e(W)$ to be the set of global states by applying e to the states in W. Then, V contains a bivalent state.

aka if G is bivalent, we can apply any e to it. but, we delay this e and apply it to some $G2$ reachable from G. There exists a $e(G2)$ that is still bivalent. ($G = G2$ is ok)

Proof: Assume V does not contain such a $G2$

Claim 1: There must exist some schedule σ such that σ contains the event e and $\sigma(G)$ is 0-valent.

- G is bivalent, so there must exist some 0-valent $G0$ reachable from G where $G0 = \sigma 1(G)$.
- Consider:
  - $\sigma 1$ contains e: we are done
  - $\sigma 1$ does not contain e: just append e to $\sigma 1$ to get σ

Claim 2: There must be a 0-valent state $G0$ in V.

- From Claim 1: $\sigma(G)$ is 0-valent
- We know e exists in σ. Remove events from the head of σ until the head is e. Let this global state be $G0$.
- Since we claim there are no bivalent states in V, $G0$ is either 0-valent or 1-valent. It must be 0-valent because we reached a 0-valent state from it.

Claim 3: There must be a 1-valent state $G1$ in V.

- Symmetric to claim 2

Claim 4: There must be $F0$ and $F1$ in W such that $e(F0) = G0$ is 0-valent, $e(F1) = G1$ is 1-valent, and $F0$ and $F1$ are neighbours separated by an event d.

- Show such neighbours exist through inductive reasoning
- e and d must occur on the same process p, otherwise $G1 = e(F1) = e(d(F0)) = d(G0)$ will have a decision of 0. (by Lemma 2 - events are swappable if processes are disjoint)
- Intuition: the ordering of e and d is entirely responsible for the system deciding on 0 or 1. Force p to be slow such that the other processes will decide before p executes either e or d.
- ...
- We find that we can reach a 1-valent state after reaching a 0-valent state.
- Thus, contradiction

Actual Proof Sketch

- Start with some initial bivalent state (by Lemma 1)
- Let processes take steps in round-robin fashion. For some process p's turn:
- If the message system contains no message, return null

- otherwise, return the oldest message
- let G be the current state
- execute (p, m) if $e(G)$ is bivalent
- otherwise, find a finite length σ that does not contain e and $e(\sigma(G))$ is bivalent (by Lemma 4)
- apply σ and e
- the system is always bivalent

Version 5: Byzantine Failures; Reliable Channels; Synchronous

- **Termination**: all non-faulty nodes eventually decide
  - proof: trivial because $f + 1$ rounds
- **Agreement**: all non-faulty nodes should decide on the same value
  - proof:
    - at least one phase is deciding
    - after that deciding phase, all non-faulty processes have the same $V[i]$ (by Lemma 1)
    - in the following phases, $V[i]$ never changes on the non-faulty processes (by Lemma 2)
- **Validity**: if all non-faulty nodes have the same initial input: that value is the only possible decision
  - to avoid a trivial algorithm
  - must ignore Byzantine nodes' inputs because they can ignore / "change" their input
  - proof: follows from Lemma 1

Main problem: even if you can detect that some node is faulty, hard to decide which node is faulty

> Theorem: If $n \leq 3f$, Byzantine consensus problem cannot be solved. (proof outside of CS4231)

THE PROTOCOL

- Every protocol takes turns being the coordinator in *rounds*
- If a coordinator is non-faulty, all processes will see the proposal and achieve consensus
- A phase is **deciding** if the coordinator is non-faulty
  - Need to ensure that after a deciding phase, the decision doesn't change

```
// process i, with my_input as input
Consensus(i, my_input) {
    V[1..n] = 0
    V[i] = my_input

    // f+1 phases
    for k = 1; k ≤ f+1; k++ {
        // all-to-all broadcast
        send V[i] to all processes
        set V[1..n] to the n values received

        if X occurs (> n/2) times in V {
            // majority
            proposal = X
        } else {
            proposal = 0
        }

        // coordinator
        if (k == i) {
            // i am coordinator
            send proposal to all
        } else {
            receive the coordinator's proposal
        }

        // should I listen to coordinator?
        if value y occurs (> n/2 + f) times in V {
            // *overwhelming* majority -> ignore coordinator
            V[i] = y
        } else {
```

```
            V[i] = coordinator's proposal
        }
    }

    decide on V[i]
}
```

Will always have a majority if n is odd (because the domain is 0 and 1)!

- if a message is not received (because synchronous + reliable):
  - the sender must be faulty, so just set value to 0 or 1 arbitrarily (doesn't matter because the node is faulty anyways)

The Protocol Proof

Lemma 1

> Lemma 1: if all non-faulty processes P_i have $V[i] = y$ at the beginning of phase k, then this remains true at the end of phase k.

This is true because of the overwhelming majority rule.

$$\begin{aligned} & n - f > n/2 + f \\ \iff & n/2 > 2f \\ \iff & n > 4f \text{ (true by definition)} \end{aligned}$$

Lemma 2

> Lemma 2: if the coordinator in phase k is non-faulty, then all non-faulty processes P_i have the same $V[i]$ at the end of phase k.

tl;dr: X is not majority in coordinator => X is not overwhelming majority in any other node

Case 1: coordinator received a value X that is the majority

- If some value Y is the overwhelming majority, then X == y
  - because of simple math (X is already at least half, another value cannot also be the majority)
- Otherwise, non-faulty nodes will take the coordinator's proposal

Case 2: coordinator did not receive a majority (proposal = 0)

- No value is the majority => no value is the overwhelming majority
- All non-faulty nodes will take the coordinator's proposal

Self Stabilisation

Distributed systems can be represented by a graph, but we typically only need a (ideally minimum) spanning tree to know how to send messages. The actual graph can change over time due to faults: topology changes, failures, reboots. This can cause the spanning tree to enter an *illegal* state.

Self Stabilising Spanning Tree Algorithm

Specs:

- constructs a spanning tree, rooted at a special node
- can be used to compute shortest path
- runs in the background, constantly updating the variables
  - will eventually be correct (might be wrong an any point in time)
  - can be overwhelmed if there are too many failures

Algorithm:

- each process maintains two variables: `parent`, `dist` (distance to root)
- on root node (executed periodically):
  - sets `dist = 0`; `parent = null`
- on all other nodes (executed periodically):
  - retrieve `dist` from neighbours into `neighbour_dists`
  - set `dist = min(neighbour_dist) + 1`
  - set `parent = neighbour (with smallest dist)` - tie break arbitrarily
- ^these steps do **not** need to be executed atomically (arbitrary interleaving is allowed)

Proof

Phase: minimum time period where every process has executed its code at least once ("has taken an action")

- `A_i` (aka level): actual shortest distance from process `i` to root
  - a node at level x has at least one neighbour in level $x-1$
  - a node at level x only has neighbours in level $x-1$, x, $x+1$
- `dist_i`: currently known shortest distance from process `i` to root

(let process 1 be the root)

Lemma:

- at the end of phase 1, `dist_i = 0`, `dist_i ⩾ 1` for any `i ⩾ 2`
- at the end of phase r,
  - any process i where $A_i \leq r-1$ has $dist_i = A_i$ (actual distance found)
  - any process i where $A_i \geq r-1$ has $dist_i \geq r$

Inductively prove the following for phase r+1:
(common prove technique for self-stabilising algorithms)

1. prove that the actions will not roll back what is already achieved by phase r (no regression)
   - claim: the "already know" conditions *hold throughout* phase r+1
     - hold throughout is stronger than holds through at the end!!
     - proof: prove all cases using neighbour levels being one-away property
2. prove that at some point, each node will achieve more (has progress)
   - claim: the "want to show" conditions *holds through at some point* in phase r+1
     - proof: prove all cases using "action must happen" + "already know" conditions + neighbour-level property
3. prove that no regression happens in this phase for the progress made in this phase
   - because multiple actions are done in parallel (no serialisation of actions)
   - claim: for all nodes, after the "want to show" conditions holds through, it must continue to hold through until the end of phase r+1
     - proof: simple to show regression is not possible

Nodes with	already know: phase r	want to show: phase r+1
$A_i \leq r-1$	$dist_i = A_i$	$dist_i = A_i$
$A_i = r$	$dist_i \geq r$	$dist_i = A_i$
$A_i \geq r+1$	$dist_i \geq r$	$dist_i \geq r+1$

- **Theorem 1**: After $H+1$ phases, `dist_i = A_i` on all nodes
  - where H is `max(A_i)`
  - directly from Lemma
- **Theorem 2**: After $H+1$ phases, `dist` and `parent` on all nodes are correct
  - proof: all nodes (except root) has a single parent pointer => n nodes, n-1 edges; all nodes have a path to root => connected; therefore, it is a spanning tree