CS4231 — PARALLEL AND DISTRIBUTED ALGORITHMS

PROF. YU HAIFENG

arsatis[*]

CONTENTS

[*]author: https://github.com/arsatis

1 MUTUAL EXCLUSION

Lecture 1
11th January 2023

In shared memory systems, the use of multi-threading or multi-processing may introduce several unexpected results.

process 0	*process 1*
read x into a register (value read: 0)	
increment the register (1)	
	read x into a register (value read: 0)
	increment the register (1)
write value in register back to x (1)	
	write value in register back to x (1)

Figure 1: When two processes increment x concurrently, we would expect the end result to be $x = 2$, but $x = 1$ instead in this particular case.

To avoid erroneous behaviour in parallel systems, we can utilize **mutual exclusion** to synchronize access to shared resources by implementing **critical sections (CS)** with the following properties:

1. **Mutual exclusion**: there should be ≤ 1 process in the CS at any time.
2. **Progress**: if some process P_1 is not in the CS, then P_1 cannot prevent some other process P_2 from entering the CS.
3. **No starvation**: if some process P is waiting on the CS, it will eventually enter it.
    - We can see that no starvation implies progress (i.e., the former is a stronger property than the latter).

1.1 *Peterson's algorithm*

```
1:  int turn = 0;
2:  bool wantCS[2] = {false, false};
3:
4:  function RequestCS(x)                          ▷ x ∈ {0, 1}
5:      wantCS[x] = true;
6:      turn = 1 − x;
7:      while wantCS[1 − x] == true && turn == 1 − x do
8:
9:  function ReleaseCS(x)
10:     wantCS[x] = false;
```

1.1.1 Mutual exclusion

Mutual exclusion: proof by contradiction, assume both P_0, P_1 are in the CS.

1. Case 1: turn == 0.
   (a) Then, it must be the case that P_0 executed turn = 1 before P_1 executed turn = 0.
   (b) Since P_1 is in the CS, it must be the case that wantCS[0] == false as seen by P_1.
   (c) However, P_0 has executed line 6 before P_1, thus wantCS[0] == true as seen by P_1 when it is at line 7 (and kept true, since P_0 is still in the CS). This is a contradiction.
2. Case 2: turn == 1. The argument is symmetrical to case 1.

1.1.2 Progress

Progress: there are 3 cases to consider:

1. No process in CS, P1 wants to enter.
2. No process in CS, P2 wants to enter.
3. No process in CS, both P1 and P2 want to enter.

We will prove case 3, and consider the value of "turn" when both P_0 and P_1 are waiting (i.e., wantCS[0] == true and wantCS[1] == true).

1. Case 1: turn == 0. Then P_0 can enter (progress is made).
2. Case 2: turn == 1. Then P_1 can enter.

1.1.3 Starvation

No starvation: assume that processes never enter the CS.

1. Case 1: P_0 is waiting.
   (a) Then, it must be that wantCS[1] == true and turn == 1.
   (b) Since wantCS[1] == true, P_1 must be somewhere between lines 4 and 6. Since turn == 1, P_1 will eventually enter and exit the CS, setting wantCS[1] = false.
   (c) If P_1 enters again immediately, it will set wantCS[1] = true and turn = 0, before getting stuck on the while loop. P_0 will eventually see these updates and fall through the while loop.
2. Case 2: P_0 is waiting. The argument is symmetrical to case 1.

1.2 *Lamport's bakery algorithm*

Intuition: each process will get a number, and be served when all processes with a lower number have been served.

```
1:  int number[n] = {};                                   ▷ filled with 0
2:  bool choosing[n] = {};                                ▷ filled with false
3:
4:  function Smaller(id_1, id_2)                          ▷ utility function
5:      if number[id_1] == number[id_2] then
6:          return id_1 < id_2;
7:      return number[id_1] < number[id_2];
8:
9:  function RequestCS(id)
10:     choosing[id] = true;                              ▷ get a number
11:     for j = 0; j < n; j++ do
12:         tmp = number[j]
13:         if tmp > number[id] then
14:             number[id] = tmp;
15:     number[id]++;
16:     choosing[id] = false;
17:
18:     for j = 0; j < n; j++ do      ▷ wait for processes with a lower number
19:         while choosing[j] == true do
20:         while number[j] != 0 && Smaller(j, id) do
21:
22: function ReleaseCS(id)
23:     number[id] = 0;
```

1.2.1 Mutual exclusion

Mutual exclusion: suppose two processes x, y are in the CS, and wlog, assume Smaller(x,y) when they are in the CS.

1. Then, y must see number[x] == 0 when it is at line 19 (or else, it wouldn't be able to enter the CS). Let this time be T_1. This is only possible when:
   (a) Case 1: x has not executed line 12 for $j = y$.
      i. This means that eventually, number[x] > number[y] before x enters the CS. This contradicts our initial assumption.
   (b) Case 2: x has executed line 12.
      i. However, x should not have executed line 14, because this contradicts the fact that number[x] == 0.
      ii. Therefore, x must be at line 13 when y is at line 19.

2. Next, we consider the time T_2 when y invoked line 18 for $j = x$ and passed that statement.
   (a) Case 1: x has finished choosing and has executed line 15, but this is impossible since $T_2 < T_1$ for process y, but $T_2 > T_1$ for process x.
   (b) Case 2: x has not started choosing and has not executed line 10. However, this means that eventually number$[x]$ > number$[y]$, which is a contradiction.
3. Therefore, both processes x, y cannot both be in the CS.

1.2.2 Progress

Progress: assume that there is no progress, and let process x have the smallest queue number.

1. Case 1: x is blocked at line 18.
   (a) However, process j will eventually set choosing$[j]$ = false and block before the CS (since we assumed that there is no progress).
   (b) Therefore, choosing$[j]$ == true as seen by x, and x will eventually make progress.
2. Case 2: x is blocked at line 19.
   (a) However, x has the smallest queue number, so it cannot block here.
3. Therefore, there is progress for process x.

1.2.3 Starvation

No starvation: assume that processes never enter the CS, and let process x have the smallest queue number.

1. Case 1: x is blocked at line 18.
   (a) However, choosing$[j]$ will eventually be true within a finite amount of time, as seen above. Thus x cannot be blocked here.
2. Case 2: x is blocked at line 19.
   (a) However, x has the smallest queue number, so it cannot block here.
3. Therefore, process x does not experience starvation.

1.3 *Hardware solutions*

1. **Disable interrupts**: disallowing context switches would prevent processes from reading stale values.
2. **Special machine-level instructions**: *atomic instructions* (e.g., TestAndSet) can prevent updates from getting lost.

1.4 *Remark: correctness proofs for mutual exclusion*

We make the following assumptions throughout our proofs and discussions:

- A process in the CS will eventually leave it.
- Processes will always continue to execute their instructions (i.e., there is no concept of "priority" between processes leading to starvation).

When writing proofs:

- We always need to consider the worst-case schedule.
- We need to specify the *time* in our proofs, e.g.:
  - P_0 and P_1 are both in the CS, or are waiting to enter the CS.
  - Consider various cases to narrow down the particular line(s) which one of the processes is currently executing, etc.
- We need to consider the case whereby the other process in consideration completes and starts again immediately, before the current process is able to respond (especially for proofs on *no starvation*).
- We should only analyze where processes may be stuck on a case-by-case basis if `while` loops are arranged **sequentially**; if the `while` loops are **nested**, it is not well-defined where processes may get stuck.

2 SYNCHRONIZATION PRIMITIVES

Lecture 2
18th January 2023

Synchronization primitives (e.g., semaphores, monitors, locks, conditional variables, etc.) are OS-level APIs which the program may call to synchronize processes without any busy waiting.

2.1 *Semaphores*

Semaphores are equipped with the following atomic operations:

```
1: bool value = true;
2: queue<Process> queue;      ▷ queue of blocked processes, initially empty
3:
4: function P( )
5:     if value == false then
6:         add myself to queue
7:         block                    ▷ if blocks, context switch to other process
8:     value = false;
9:
```

```
10: function V( )
11:     value = true;
12:     if queue is not empty then
13:         wake up an arbitrary process in queue    ▷ wakes exactly 1 process
```

The queue is not necessarily FIFO; depends on the implementation.

Then, RequestCS(x) can be implemented by simply making a call to P( ), whereas ReleaseCS(x) can be implemented simply using V( ).

2.2 *Monitors*

A **monitor** is an object (activated using `synchronized()` in Java) with the following properties:

- A **monitor queue**: serves as a lock when processes enter and exit a monitor.
  - enterMonitor( ): Enters if no one is inside the monitor, else it adds itself to the *monitor queue* and blocks.
  - exitMonitor( ): Picks an arbitrary process from the *monitor queue* and unblocks it, if the *monitor queue* is not empty.
- A **wait queue**: when inside a monitor.
  - object.wait( ): Atomically (1) adds itself to the *wait queue*, (2) exits the monitor (i.e., releases the monitor lock), and (3) block.
  - object.notify( ): Picks an arbitrary process from the *wait queue* and unblocks it, if the *wait queue* is not empty.
  - object.notifyAll( ): Unblocks all processes in the *wait queue* if it is not empty.

Note that object.wait( ) will only release the inner monitor's lock in Java, if it is called within a nested monitor.

The notification is lost if no process is waiting on the monitor.

There are two types of monitors:

1. **Hoare-style monitor**: signaler gives lock & CPU to waiter; waiter runs immediately and gives up lock & CPU back to signaler when it exits CS or if it waits again.
2. **Mesa-style monitor**: signaler keeps lock & CPU, waiter placed on monitor queue with no special priority.

a.k.a. Java-style monitor.

3 CONSISTENCY CONDITIONS

Lecture 3
25th January 2023

Consistency specifies what behaviour is allowed when a shared object is accessed by multiple processes. When we say something is consistent, it means that it satisfies the specification (according to some given specification).

To formalize our discussion on consistency, we introduce the following terms:

- **Operation** (e): a single *invocation/response pair* of a method of a shared object, by a process.
  - **proc**(e) rep. the invoking process.
  - **obj**(e) rep. the object containing the method.
  - **inv**(e) rep. the *invocation event* (i.e., start time).
    - Two invocation events are the same if the invoker, invokee, and parameters (i.e., inv(p, read, X)) are the same.
  - **resp**(e) rep. the *reply event* (i.e., end time).
    - Two response events are the same if the invoker, invokee, parameters, and response (i.e., resp(p, read, X, 1)) are the same.
- **(Execution) history** (H): a sequence of invocations and responses ordered by wall clock time.
  - A history H is **sequential** if any invocation is always *immediately followed by* its response (i.e., no interleaving among executions).
  - A sequential history H is **legal** if all responses satisfy the *sequential semantics* of the data type.
    - **Sequential semantics**: the semantics/logic obtained if there is only one process accessing the data type.
  - Two histories are **equivalent** if they have the same set of events (i.e., ordering of events may be different, but all responses must be the same).
- **Process subhistory**: process p's process subhistory of H (i.e., H|p) is the subsequence of all events of p.
- **Object subhistory**: object o's object subhistory of H (i.e., H|o) is the subsequence of all events of o.
- **Process order**: a partial order among all events.
- **External order**: for two events e_1, e_2, $e_1 < e_2$ iff the resp(e_1) appears in H before inv(e_2).

For any *invocation* in H, the corresponding response is required to be in H.
Else, it is a *concurrent history*.

A process subhistory is always sequential.

For any two events of the same process, process order is the same as execution order.

3.1 *Sequential consistency*

A history H is **sequentially consistent** if it is *equivalent* to some *legal sequential history* S that *preserves process order*.

3.2 *Linearizability*

There are two definitions for linearizability:

1. A history is **linearizable** if it is *equivalent* to some execution such that each operation happens *instantaneously* at some point (i.e., *linearization point*) between the invocation and response.

2. A history H is **linearizable** if:
   (a) it is *equivalent* to some *legal sequential history* S, and
   (b) S preserves the external order in H (i.e., the partial order induced by H is a subset of the partial order induced by S).

Linearizability is a **local property**, i.e.,

H is linearizable iff for every object x, H|x is linearizable.

Note that *sequential consistency* is *not* a local property.

3.3 *Consistency definitions for registers*

Registers are abstract data types, where a single value can be read or written at any point in time. We can define an implementation of a register as:

- **Atomic** if the implementation always ensures *linearizability* of the history.
- **Sequentially consistent** if the implementation always ensures *sequential consistency* of the history.
- **Regular** if the implementation always ensures that:
  1. When a read does not overlap with any write, the read returns the value written by *one of the* ***most recent writes***.
  2. When a read overlaps with some write(s), the read returns the value written by *one of the* ***most recent writes*** *or one of the* ***overlapping writes***.

  – Thus, atomic $\implies$ regular.
  – However, sequentially consistent registers may not be regular, and vice versa.

most recent write: a write that overlaps with the write with the latest resp(e).

- **Safe** if the implementation always ensures that:
  1. When a read does not overlap with any write, the read returns the value written by *one of the* ***most recent writes***.
  2. When a read overlaps with some write(s), the read can return **anything**.

  – Thus, regular $\implies$ safe.
  – However, sequentially consistent registers may not be safe, and vice versa.

4 MODELS & CLOCKS

Lecture 4
1st February 2023

In this chapter, we assume that processes can perform three kinds of atomic actions/events (relative to the distributed system):

- **Local computation**.
- **Send** a single message to a single process.
- **Receive** a single message from a single process.

We also assume that the communication model is:

- Point-to-point.
- Error-free, with an infinite buffer.
- Messages received can be potentially out of order.

Software clocks allow distributed computing protocols to infer ordering among events, and they incur much lower overhead compared to maintaining physical clocks. Event orderings which are captured by software clocks include:

- **Process order**: e.g., $A \rightarrow B$ within the *same thread*.
- **Send-receive order**: e.g., $B \rightarrow C$ across *different threads*.
- **Transitivity**: $(A \rightarrow B) \wedge (B \rightarrow C) \implies (A \rightarrow C)$.

Specifically, the goal of software clocks is to capture *"happens before"* relationships, which is a partial order among events. We will explore 3 different types of software clocks.

4.1 *Logical clocks*

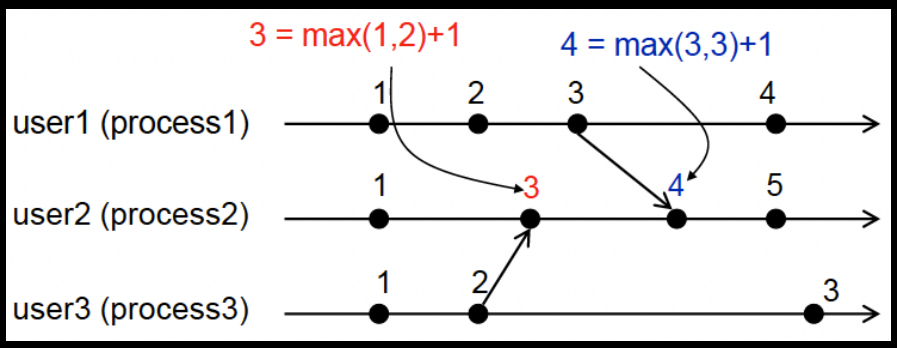
In **logical clocks**, each event has a single integer as its logical clock value. Specifically,

1. Each process has a local counter C.
2. Increment C at each "local computation" and "send" event.
3. When sending a message, the current local counter value V is attached to the message.
4. At each receive event, the receiving process sets $C = \max(C, V) + 1$.

Note that +1 is just by convention, the logical clock still works if we increment it with any random positive value.

Theorem (*logical clocks*):

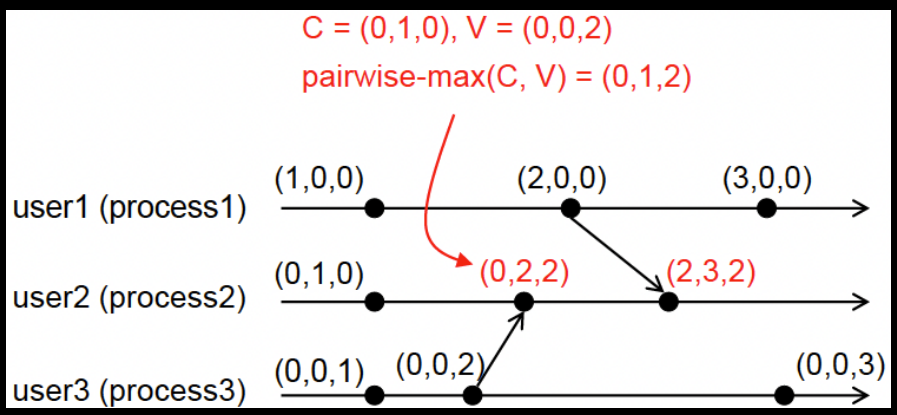
Event s happens before $t \Rightarrow$ logical clock value of $s <$ logical clock value of t.



4.2 *Vector clocks*

In **vector clocks**,

1. Each process i has a local vector C.
   - The i^{th} entry in C is called the **principle entry** of process i (i.e., only process i is able to modify that entry).
2. Increment C[i] at each "local computation" and "send" event.
3. When sending a message, the local vector clock value V is attached to the message.
4. At each receive event, C = pairwise-max(C, V), C[i] + +.



Theorem (*vector clocks*):

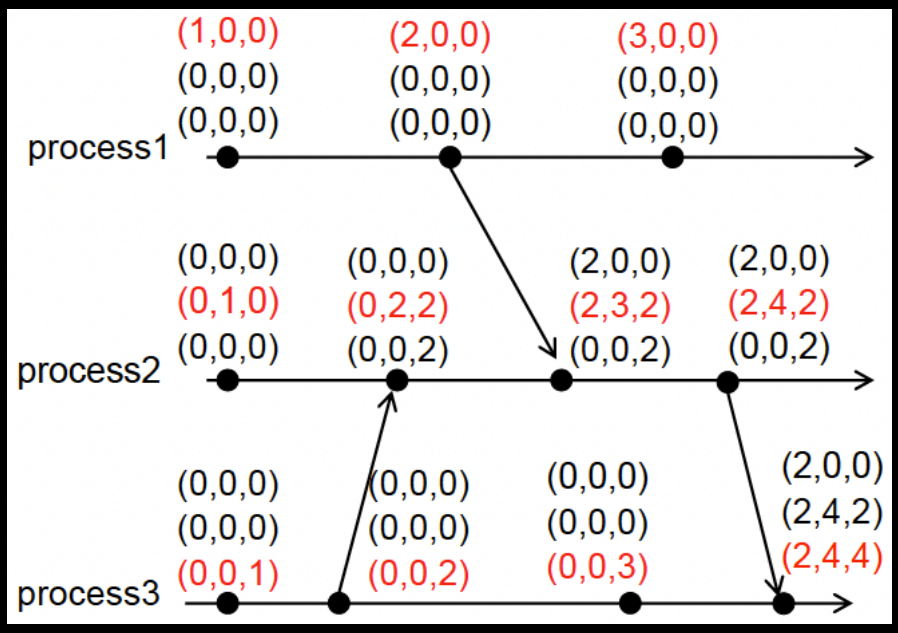
Event s happens before $t \Leftrightarrow$ vector clock value of $s <$ vector clock value of t.

Unlike events in logical clocks, there is no total ordering among events in vector clocks (e.g., $< 1, 2 >$ vs $< 2, 1 >$).

4.3 *Matrix clocks*

In **matrix clocks**,

1. Each event has n vector clocks, one for each process.
   - The i^{th} vector on process i is called the **principle vector** of process i.
   - Principle vectors serve the same function as the vectors in vector clocks.
   - Non-principle vectors are just piggybacked on messages to update knowledge regarding what other processes have seen.
2. For principle vector C,
   (a) Increment C[i] at each "local computation" and "send" event.
   (b) When sending a message, all n vectors are attached to the message.
   (c) At each "receive" event, C = pairwise-max(C, V), C[i] + +.
3. For non-principle vectors C′, C′ = pairwise-max(C, V) (i.e., no incrementing of the principle entry).



Matrix clocks describe what other processes have seen, which can be beneficial to some applications (e.g., garbage collection).

5 GLOBAL SNAPSHOTS

Lecture 5
8th February 2023

It is useful to be able to take a snapshot of the global computation (i.e., a snapshot of local states on all n processes), for:

- Debugging.
- Backup and checkpointing (i.e., for restoring the state of the current snapshot in the event of a crash).
- Calculating some global predicate (e.g., total currency in a country).

in the past: we don't know when it occurred.
could have happened: the snapshot may not have happened in the past, but the user is unable to tell the difference.

Formally, we define the following:

- **Consistent snapshot**: a snapshot of local states on n processes, such that the *global snapshot could have happened some time in the past*.
- **Global snapshot**: a set of events such that if e_2 is in the set and e_1 **is before** e_2 **in process order**, then e_1 must be in the set.
- **Consistent global snapshot (CGS)**: a **global snapshot** such that if e_2 is in the set and e_1 **is before** e_2 **in send-receive order**, then e_1 must be in the set.

Note that *transitivity* is implicit in the definition of CGS.

Theorem (*consistent global snapshots*):

Consider the ordered events $e_1, e_2, \ldots$ on any process. $\forall m \in \mathbb{Z}^+$, there exists a consistent global snapshot S such that $\begin{cases} e_i \in \mathrm{S} & \text{for } i \leq m \\ e_i \notin \mathrm{S} & \text{for } i > m \end{cases}$

5.1 *CGS protocol*

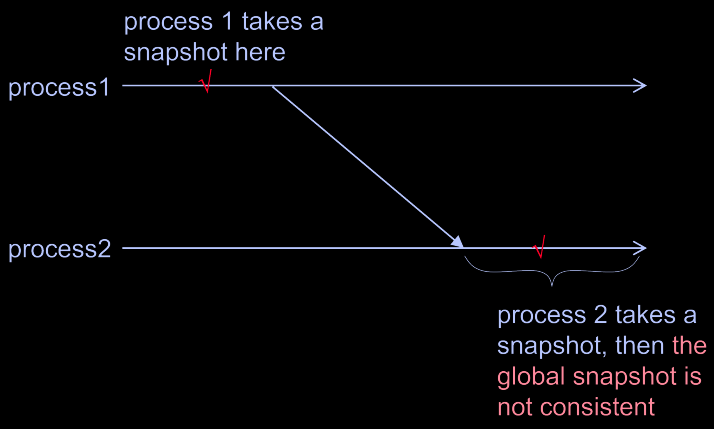
We make the following assumptions on the communication model when discussing protocols for CGS:

- There is no message loss.
- Communication channels are unidirectional.
- FIFO delivery on each channel (i.e., if process A sends two messages M_1 followed by M_2 to process B, B is guaranteed to always receive M_1 before M_2).

The key intuition behind a protocol that is capable of capturing a CGS is to exhaustively consider all possible cases where local snapshots could be taken. Assuming process A is sending a message to process B,

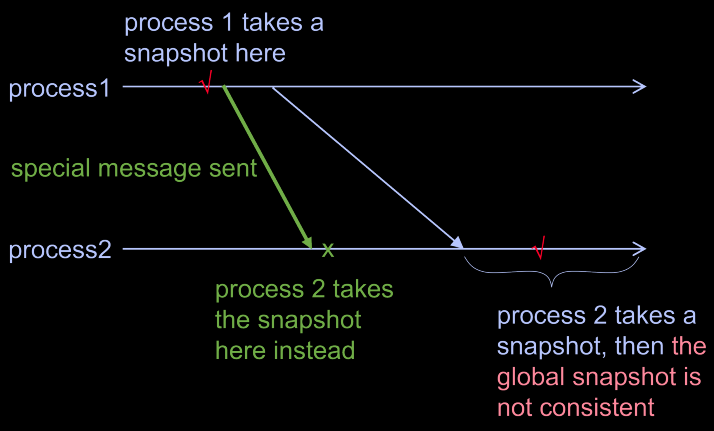
1. A takes a snapshot before send, B takes a snapshot before receive.
2. A takes a snapshot before send, B takes a snapshot after receive.
3. A takes a snapshot after send, B takes a snapshot before receive (assume that the message on the network will also be captured by the snapshot).
4. A takes a snapshot after send, B takes a snapshot after receive.

It is clear that case 2 is the only problematic case: suppose that A is sending some money to B, this amount will be doubly-recorded by the local snapshots.



The following steps describe one possible solution:

1. A will take a snapshot, then send a special message (used only for the snapshot protocol) to B before performing other actions.
2. B will take a snapshot upon receiving the special message, before performing other actions (e.g., processing the normal message sent by A).



This would enable the system to avoid the aforementioned problematic case, and thus ensure that the global snapshot is consistent.

5.2 *Chandy & Lamport's protocol*

In Chandy & Lamport's protocol, each process is either:

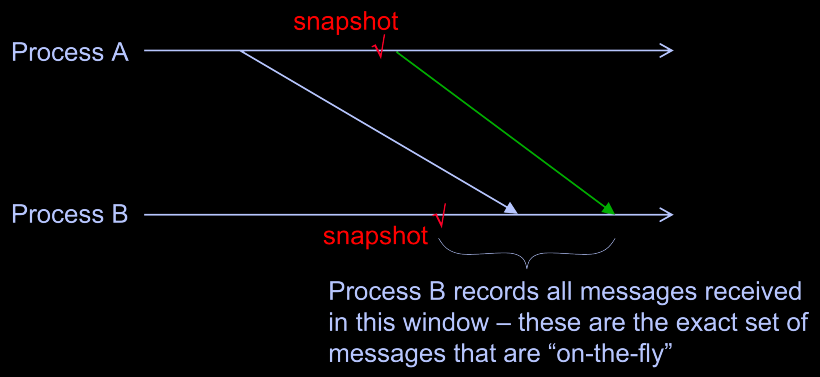
- Red (i.e., it has taken the local snapshot), or
- White (i.e., it has not taken the local snapshot).

The protocol is described as follows:

1. A single process turns itself from white to red to initiate the protocol.
2. Once a process turns red, it immediately sends out marker messages to all other processes.
3. Upon receiving a marker, the process turns red (and repeats step 2).
4. When a process receives $n-1$ marker messages, this implies that all processes have taken their local snapshot.

To capture messages that are "on-the-fly" when the snapshot is taken, processes will record all messages received in the window:

$$\{\text{time of local snapshot} \rightarrow \text{time when } (n-1)^{\text{th}} \text{ marker is received}\}$$



This enables the system to correctly rollback after a crash happens, since messages "on-the-fly" while the global snapshot is taken are captured.

6 MESSAGE ORDERING

Lecture 6
15th February 2023

Causal order: if s_1 happens before s_2, and r_1, r_2 are on the same process, then r_1 must be before r_2.

To ensure causal ordering, some possible protocols include:

- Piggybacking preceding outgoing messages on the current message, and process piggybacked messages before the actual message.
- Withholding messages which arrive out-of-order, until all preceding messages have arrived and are processed.

The following is a protocol which ensures causal ordering using matrices:

1. Each process maintains an $n \times n$ matrix M, which records how many messages are there between each pair of nodes.

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

2. When process i sends a message to process j,
   (a) $M[i, j] + +$ on process i.
   (b) Piggyback M on the message.
3. When process j receives a message from process i with matrix T piggybacked,
   - j processes the message and sets M = pairwise-max(M, T) if:
     (a) $\forall k \neq i, T[k, j] \leq M[k, j]$, and
       – These cells keep track of how many send events there are before the send event of this particular message.
       – If $\exists k \neq i, T[k, j] > M[k, j]$, this means that j should have received another message from k first before processing this message.
     (b) $T[i, j] = M[i, j] + 1$.
       – This implies that process j has received all previous $M[i, j]$ messages from i, and is thus ready to receive the $(n + 1)^{\text{th}}$ message from i.

     where M rep. the local matrix on process j.
   - Otherwise, j delays the processing of the message.

6.1 *Broadcast messages*

In **broadcast messaging**, every message is sent to all processes (including the sender itself). We may want to ensure certain orderings among messages:

- **Causal order**: similar to point-to-point messaging.
- **Total order**: all messages are delivered to all processes in the same order (i.e., **atomic broadcast**).

Note that there is no relationship between total ordering and causal ordering (i.e., neither implies the other).

To ensure causal ordering in broadcast messages, each process can simply utilize the previous protocol. To ensure total ordering, some protocols include:

- Assigning a special process as the **coordinator**; to broadcast a message,
  1. Process sends a message to the coordinator.
  2. Coordinator assigns a sequence number to the message.
  3. Coordinator forwards the message to all processes with the sequence number.
  4. Messages are processed in the order of increasing sequence number.

However, in this protocol, the coordinator has too much control over the system.

- **Skeen's algorithm**:
  - Each process maintains a logical clock, and a message buffer for unprocessed messages.
  - Broadcasting involves the following steps:
    1. Sender broadcasts its message to other processes, and places the message in their message buffers.
    2. Other processes acknowledge and reply with their current logical clock value.
    3. Sender picks the maximum over the logical clock values of all processes, and uses it as the message number. The message number is sent to receiving processes.
  - A message in the buffer is processed/removed if:
    1. All messages in the buffer have been assigned numbers, and
    2. This message has the smallest number.

6.2 *Remark: correctness proofs for message ordering protocols*

To prove that a message ordering protocol is correct, we need to prove:

1. **All messages will be delivered**.
   - e.g., by showing that the message with the earliest send event can be delivered, thus by induction, all messages will eventually be delivered.
2. **Messages will be delivered in the correct order**.
   - The definition of "correct order" depends on the ordering specified.

7 LEADER ELECTION

Lecture 7
8th March 2023

We often need a coordinator in distributed systems. With a leader, some problems can be trivially solved, e.g., mutual exclusion and total order broadcast.

7.1 *Leader election on anonymous ring*

On an anonymous ring, there are no unique identifiers. It can be shown that leader election is impossible on an anonymous ring when deterministic algorithms are used.

Proof By contradiction, assume that there is some deterministic algorithm that can solve leader election. Assume that every node has the same initial state before the start of the execution.

1. Let every node execute the first instruction in the algorithm at the same time.
2. After executing the very first instruction, all nodes will end up at the same state.
3. After executing all instructions, the state of all nodes will be the same when the algorithm terminates.
4. This means that if a node declares itself as the leader, then every node will declare itself as a leader, which gives a problematic execution.

7.2 *Leader election on ring (Chang-Roberts algorithm)*

We assume that:

- Each node has a unique identifier.
- Nodes only send messages clockwise.
- Each node acts on its own.

The Chang-Roberts protocol is described as follows:

1. A node sends an election message with its own id clockwise.
2. When a node receives an election message,
   (a) if the id in the message is larger than its own id, the message is forwarded.
   (b) if the id in the message equals its id, the node becomes the leader.
   (c) otherwise, the message is discarded.

Intuitively, it is clear that only the node with the largest id in the ring can become the leader.

- Best case runtime: $2n - 1 = O(n)$, when the election is initiated by the node with the largest id.
- Worst case runtime: $\frac{n(n+1)}{2} = O(n^2)$, when the nodes are arranged clockwise in descending order, and each node tries to send an election message with its own id.
- Average case runtime: $O(n \log n)$. Proof:
  - Let random variable x_k rep. number of messages caused by the election message from node k. Want to find: $E[\sum_k x_k] = \sum_k E[x_k]$.

By linearity of expectation.

Claim 1 $\Pr\{x_k = 1\} = \frac{n-k}{n-1}$. Proof: $n - k$ nodes have ids larger than k.

Claim 2 $\Pr\{x_k = 2 \mid x_k > 1\} = \frac{n-k}{n-2}$. Proof: $n - 2$ nodes left, $n - k$ have ids $> k$.

Claim 3 $\Pr\{x_k = k \mid x_k > k - 1\} = \frac{n-k}{n-k} = 1$. Proof: same as Claim 2.

Claim 4 Define $p = \frac{n-k}{n-1}$. Then $\forall i < k, \Pr\{x_k = i+1 \mid x_k > i\} \geq p$.

Proof By induction on i, using Claim 1, 2 and 3.

– Let random variable y rep. the number of tickets we buy until we win the lottery for the very first time. Let $Pr\{y = i+1 \mid y > i\} = p$, and $\mathrm{E}[y] = \frac{1}{p}$.

Since $\forall i < k, \Pr\{x_k = i+1 \mid x_k > i\} \geq \Pr\{y = i+1 \mid y > i\}$ (Claim 4), $\mathrm{E}[x_k] \leq \mathrm{E}[y] = \frac{1}{p} = \frac{n-1}{n-k}$. Thus,

$$\begin{aligned}
\sum_{k=1}^{n} \mathrm{E}[x_k] &= n + \sum_{k=1}^{n-1} \mathrm{E}[x_k] \\
&\leq n + \sum_{k=1}^{n-1} \frac{n-1}{n-k} && \text{from above} \\
&= n + (n-1) \sum_{k=1}^{n-1} \frac{1}{n-k} && \left(\equiv n + (n-1) \sum_{k=1}^{n-1} \frac{1}{k} \right) \\
&= n + (n-1)\mathrm{O}(\log n) \\
&= \mathrm{O}(n \log n)
\end{aligned}$$

7.3 *Leader election on general graph*

For a **complete graph** with number of nodes n in the graph known, it suffices for each node to send its id to all other nodes, and after each node has received n ids, the node with the largest id will become the leader.

For a **connected graph** with n known, each node will flood its id to all other nodes (by asking its neighbours to recursively forward its id to their neighbours, and so on), and after all nodes have received n ids, the node with the largest id will become the leader.

If n is unknown, an auxiliary protocol can be used to calculate the number of nodes in the graph. The protocol establishes a **spanning tree** whose root node is the initiator of the protocol.

1. For each neighbour, the node sends a request for it to be the node's child.
2. When a node receives a request:
   (a) If it is not yet a child of another node, it accepts the request and becomes the child of the requester.
   (b) Otherwise, it declines the request.
3. Each child node repeats step 1, until the child node is a leaf node.

4. After the spanning tree is constructed, the root node sends a request for its neighbours to count the size of their subtree.
5. The request is recursively propagated until it reaches a leaf node, which returns 1. Non-leaf nodes return the sum of its neighbours' sizes + 1.

Note that spanning trees are also useful for a variety of other purposes, e.g.:

- **Broadcast**: so that the root does not need to send out n messages.
- **Aggregation**: e.g., max, min, average, sum.

8 DISTRIBUTED CONSENSUS

Lecture 8
15th March 2023

In **distributed consensus**, each node in a set of nodes has an input and want to agree on something. Our goal is to achieve distributed consensus despite (node and link) failures.

There are 5 versions of distributed consensus, distinguished from each other based on the following definitions:

- **Failure model**: specifies what components (i.e., nodes, network) can fail, and how they can fail.
  - **Node failures** include *crash failures* and *Byzantine failures*.
  - **Link failures** include message dropping.
- **Timing model**: the way a distributed model behaves with respect to time, e.g.:
  - **Synchronous**: message delay is bounded and node processing speed is guaranteed.
  - **Asynchronous**: message propagation delay may tend to infinity in case a node is retrying to establish connection with a failed node.

Crash failure: the node stops executing machine-level instructions at some point in time.

The variants of distributed consensus which will be covered include:

0. No node or link failures.
1. Node crash failures, no link failures, synchronous.
2. No node failures, channels may drop messages.
3. Node crash failures, no link failures, asynchronous.
4. Node Byzantine failures, no link failures, synchronous.

V0 is trivial; if there are no failures, nodes will definitely be able to achieve consensus (e.g., by agreeing to pick the min/max among all inputs).

In general the goal of distributed consensus is to achieve the following:

1. **Termination**: all nodes will eventually make a decision.
2. **Agreement**: all nodes that decide will decide on the same value.

Note: termination need not hold for nodes which have failed.
Note: agreement should still hold for nodes which crash after deciding.

3. **Validity**:
   - If all nodes have the same initial input, that value should be the only possible decision value.
   - Otherwise, nodes are allowed to decide on anything.

8.1 *Timing models*

In synchronous systems:

- Message delay has a known upper bound x.
- Node processing delay has a known upper bound y.
- Both x and y are given to the algorithm/protocol as input.
- We can generalize and make all processes proceed in inter-locked **rounds**, where in each round, every process:
  1. Performs some local computation.
  2. Sends one message to every other process.
  3. Receives one message from every other process.

  Messages sent during a round will be received during that round.
  - Rounds can be implemented as having a duration of:

    message propagation delay

    + local processing delay

    + max(error between any two nodes' local clock values)

Sends to a single process can be treated as sending a valid message to one process and dummy messages to all other processes.

Costly computations can be captured as spending $k > 1$ rounds.

- Accurate failure detection is possible.
  - If nodes do not receive any messages from some node by the end of the round, they can be certain that the node has failed.

8.2 *Version 1: node crash failures (synchronous)*

- Failure model:
  - Nodes may experience crash failure.
  - Communication channels are reliable.
- Timing model: synchronous.

Solution: keep forwarding the values for $f + 1$ rounds, where f rep. the upper bound on the number of failures that could occur in the system.

A protocol for this version of distributed consensus is given as follows:

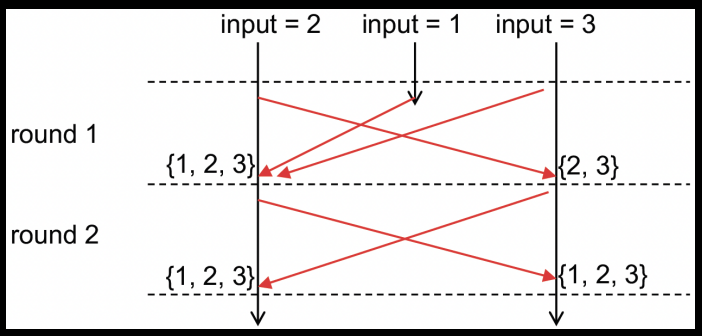


Figure 2: Sample run of the protocol with $f = 1$.

```
1: function Consensus(input)
2:     S ← {input}
3:     for i = 1 to f + 1 do
4:         send S to all nodes
5:         receive n − 1 sets
6:         for each set T received do
7:             S ← S ∪ T
8:     decide on min(S)
```

Theorem (*lower bound on number of rounds*):

With f crash failures, any consensus protocol will require $\Omega(f)$ rounds.

The proof of the theorem is not covered in the module.

Proof of correctness:

- **Termination**: trivial, since there is no wait.
- **Validity**: trivial, since S will only contain one element.
- **Agreement**:
  - We first define:
    - A node is **non-faulty** during round r if it has not crashed by round r.
    - A round is **good** if there is no node failure during that round.
  - Based on these definitions, we can make the following claims:

  Claim 1 With $f + 1$ rounds and f failures, there will be ≥ 1 good round.
  Claim 2 At the end of any good round r, all non-faulty nodes during round r will have the same S.
  Claim 3 Suppose r is a good round, and consider any round t after r. During round t, the value of S on any non-faulty nodes does not change.
  Claim 4 All non-faulty processes at round $f + 1$ will have the same S.

  Thus, by Claim 4, there is agreement.

8.3 *Version 2: link failures*

Failure model:

- Nodes do not fail.
- Communication channels may fail (i.e., may drop an arbitrary number of messages).

Timing model: synchronous.

It is impossible to achieve the goal of (1) termination, (2) agreement, and (3) validity using a *deterministic* algorithm, since the communication channel is able to drop all messages.

- To prove this, we first define the concept of **indistinguishability**:
  - Execution α is **indistinguishable** from execution β for some node, if the node sees the *same messages and inputs* in both executions.

Proof By contradiction, assume there is some deterministic algorithm which is able to achieve all these goals.

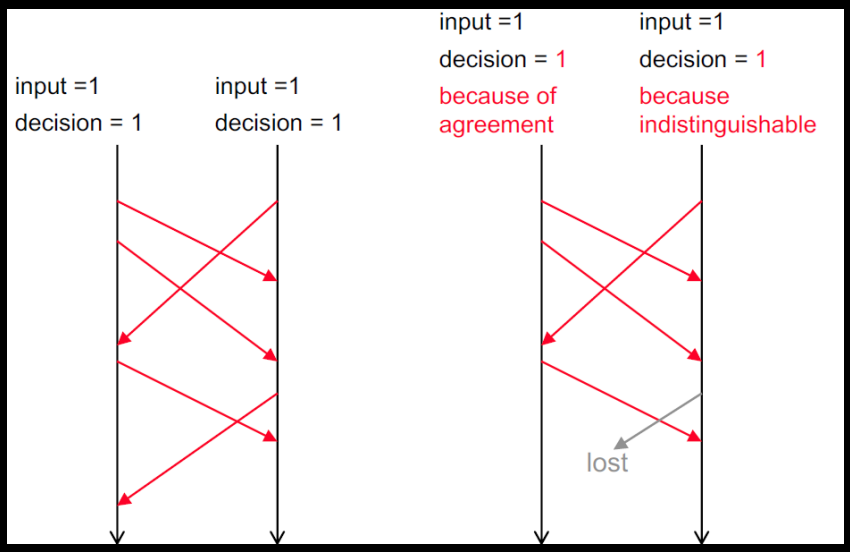
1. Let execution $\alpha = \{A : input = 0, B : input = 0\}$, and assume that all messages between A and B are dropped. Then, both nodes will arrive at the final decision of 0 (due to validity).
2. Let execution $\beta = \{A : input = 1, B : input = 0\}$, and similarly assume that all messages between A and B are dropped. Then, B arrives at the decision of 0 (due to indistinguishability), and A arrives at the final decision of 0 (due to agreement).
3. Let execution $\gamma = \{A : input = 1, B : input = 0\}$, and assume that all messages are dropped. Then, A arrives at the decision of 0 (due to indistinguishability), and B arrives at the final decision of 0 (due to agreement).
4. However, execution γ violates validity. Thus, we have arrived at a contradiction.

We can weaken the goal of validity as follows:

- **Weakened validity**: assuming the input to be either 0 or 1,
  - If all nodes start with 0, the decision should be 0.
  - If all nodes start with 1, and no message is lost throughout the execution, the decision should be 1.
  - Otherwise, nodes are allowed to decide on anything.

However, it is still impossible to achieve the weakened goal using a *deterministic* algorithm.

- Consider the following case:



1. Let execution e_1 rep. the case shown on the left. Both nodes arrive at the final decision of 1 (due to validity).
2. Remove the last message from e_1, and let this execution be e_2 (shown on the right). From the perspective of the right process, the e_2 is indistinguishable from e_1, so it reaches the decision of 1 (due to indistinguishability), and the left process reaches the decision of 1 (due to agreement).
3. We can construct a series of executions where the last message of the previous execution is dropped iteratively. Then, in the last execution α of this series of executions, all messages are dropped but the decision is still 1.
4. Let execution $\beta = \{A : input = 0, B : input = 1\}$, with all messages dropped. Then, B arrives at the decision of 1 (due to indistinguishability), and A arrives at the decision of 1 (due to agreement).
5. Let execution $\gamma = \{A : input = 0, B : input = 0\}$, with all messages dropped. Then, A arrives at the decision of 1 (due to indistinguishability), and B arrives at the decision of 1 (due to agreement).
6. However, execution γ violates validity. Thus, we have arrived at a contradiction.

We can also weaken the goal of agreement as follows:

Lecture 9
22nd March 2023

- **Weakened agreement**: all nodes will decide on the same value with probability $1 - p_{\text{error}}$.
  - More precisely, we can suppose that there is an **adversary** which tries to maximize p_{error} by e.g.:
    - Setting the inputs of the processes.
    - Causing message losses.

Adversaries can be *oblivious* or *adaptive*; here, we assume an oblivious adversary which does not know the outcomes of random coin flips beforehand.

> In reality, there is no human adversary; rather, we are trying to best the worst case scenario.

With the weakened agreement and validity goals, it is possible to solve the problem using a randomized algorithm. For simplicity, we consider the (generalizable) case where:

- There are two processes P_1 and P_2.
- The algorithm runs for a predetermined number of rounds r.
- An adversary determines which messages get lost, **before** seeing the random choices.

The protocol is described as follows:

- P_1 initially picks a random integer $bar \in [1, r]$.
- P_1 and P_2 each maintains a *level* variable, L_1 and L_2 (initially set to 0), such that upon receiving a message L_j, each process P_i sets $L_i = L_j + 1$.

  $\implies$ At the end of any round, L_1 and L_2 differ by at most 1.
- Each process's input, *bar*, and *level* variables are attached to all messages.
- At the end of r rounds,
  1. P_1 decides on 1 iff:
     - P_1 knows that P_1 and P_2's input are both 1, and
     - $L_1 \geq bar$.

     $\implies$ If P_1 does not see P_2's input, it will decide on 0.
  2. P_2 decides on 1 iff:
     - P_2 knows that P_1 and P_2's input are both 1, and
     - P_2 knows *bar*, and
     - $L_2 \geq bar$.

     $\implies$ If P_2 does not see P_1's input or *bar*, P_2 will decide on 0.

Given the above protocol, we can see that errors only occur under the following cases:

1. Case 1: P_1 sees P_2's input, but P_2 does not see P_1's input or bar.

   $\implies$ $L_1 = 1$ and $L_2 = 0$, and the error occurs when $bar = 1$.
2. Case 2: P_2 sees P_1's input and *bar*, but P_1 does not see P_2's input.

   $\implies$ $L_1 = 0$ and $L_2 = 1$, and the error occurs when $bar = 1$.
3. Case 3: P_1 sees P_2's input and P_2 sees P_1's input and *bar*.

   $\implies$ Error occurs when $bar = \max(L_1, L_2)$.

Therefore, with probability $1 - \frac{1}{r}$, P_1 and P_2 will agree on the same decision.

8.4 *Version 3: node crash failures (asynchronous)*

Failure model:

- Nodes may experience crash failure.
- Communication channels are reliable.

Timing model: asynchronous (i.e., process and message delays are finite but unbounded).

- Since delays are unbounded, a **round** is no longer well-defined.

8.4.1 FLP impossibility theorem

Given the goals of termination, agreement, and weakened validity, it is impossible to solve the distributed consensus problem under the asynchronous communication model, even with only a single node crash failure.

- The fundamental reason is: the protocol is unable to accurately detect node failure.

Definitions/Formalisms:

- We assume that:
  - Each process has some local state and two variables, $input \in \{0, 1\}$ and $decision \in \{\text{null}, 0, 1\}$.
  - Each communication channel has some state (and is non-blocking).
  - The **message system** captures the state of all communication channels, i.e., $\{(p, m) | \text{message } m \text{ is on the fly to process } p\}$.
- We define:
  - **Send**: adding a pair (dest, message) to the message system.
  - **Receive**: either
    - Removing some pair from the message system and returning the message, or
    - Leaving the message system unchanged and return null (i.e., enables simulation of out-of-order receives).
  - **Global state of system**: includes all process states and the state of the message system.
    - G is **0-valent** if 0 is the only decision reachable from G.
    - G is **1-valent** if 1 is the only decision reachable from G.
    - G is **univalent** if G is either 0-valent or 1-valent.
    - G is **bivalent** if it is not univalent (i.e., can reach any decision).

  - **Step** of a protocol: takes the system from one global state to another, by executing the following on process p:
    1. Receive a message m (which can be null).
    2. Based on p's local state and m, send an arbitrary but finite number of messages.
    3. Based on p's local state and m, change p's local state to some new state.
  - **Events**: inputs to the state machine that cause state transitions.
    * An event can only be applied to a global state if:
      · m is null, or
      · (p, m) is in the message system.
    * The **execution** of a protocol can be abstracted to be an *infinite* sequence of events → failed processes will have a finite sequence of events.
  - **Schedule** σ: sequence of events that captures the execution of some protocol.
    * σ can be applied to G if the events $e \in \sigma$ can be applied to G in the order in σ.
    * $G' = \sigma(G)$ means if we apply σ to G, we will end up with G'.
  - **Reachability**: given a consensus protocol A, a global state G_2 is **reachable** from G_1 if there is a schedule σ of A such that $G_2 = \sigma(G_1)$.

Given the above definitions, by the requirements of consensus, any protocol A that works must satisfy:

- **Agreement**: no reachable global state from any initial state has more than one decision.
- **Validity**: if all nodes have the same initial input, they should all decide on that.
- **Termination**: eventually, all processes decide.

We will prove that an adversary (which can pick which messages to deliver and which processes will take the next step) can always keep the system in a bivalent state. To proceed with the proof, we first introduce some lemmas:

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

Proof By contradiction, by considering $n + 1$ initial states $(0, 0, ..., 0)$, $(1, 0, ..., 0), ..., (1, 1, ..., 1)$. The first initial state is 0-valent and the last initial state is 1-valent, and thus there must be two adjacent initial states s_i, s_{i+1} where s_i is 0-valent and s_{i+1} is 1-valent, and these two states differ by input to a single process p. Now consider an execution starting from both states, where p fails from the very beginning.

Lemma 2 Let σ_1 and σ_2 be two schedules, such that the set of processes executing steps in σ_1 are disjoint from the set that execute steps in σ_2. Then, for any G that σ_1 and σ_2 can both be applied, we have $\sigma_1(\sigma_2(G)) = \sigma_2(\sigma_1(G))$.

Proof By induction on $k = \max(|\sigma_1|, |\sigma_2|)$.

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

Proof Trivial when m is null. If m is non-null, note that all m are distinct, so it will never be sent to process p unless e has been applied.

Lemma 4 Let G be a bivalent state, and $e = (p, m)$ be an event that can be applied to G. Let W be the set of global states that is 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.

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

Proof G is bivalent, thus we must have a 0-valent state G_0 reachable from G where $G_0 = \sigma_1(G)$. If σ_1 contains e, then $\sigma = \sigma_1$. Else, let $\sigma = e||\sigma_1$, and $\sigma(G) = e(G_0)$ is 0-valent since G_0 is 0-valent.

Claim 2 There must be a 0-valent state G_0 in V.

Proof Consider the σ as defined in Claim 1. Consider the prefix σ' of σ whose last event is e. Let $G_0 = \sigma'(G) \in V$. Because V does not contain bivalent states, and because the 0-valent state $\sigma(G)$ is reachable from G_0, G_0 must be 0-valent.

Claim 3 There must be a 1-valent state G_1 in V.

Proof Symmetrical to the proof for Claim 2.

Claim 4 There must be $F_0, F_1 \in W$ s.t. $e(F_0)$ is 0-valent, $e(F_1)$ is 1-valent, and either $F_1 = d(F_0)$ or $F_0 = d(F_1)$.

Proof Let G_0 be a 0-valent state in V, and G_1 be a 1-valent state in V. W.l.o.g., assume $e(G)$ is 0-valent. Suppose $G_1 = e(\sigma_1(G))$, with $|\sigma_1| \geq 1$. Then we can find F_0 and F_1 between G and G_1 (if $e(G)$ is 1-valent, F_0 and F_1 can be found between G and G_0).

Proof for Lemma 4 Consider $F_0, F_1 \in W$ s.t. $e(F_0) = G_0$ is 0-valent, $e(F_1) = G_1$ is 1-valent, and w.l.o.g. assume $F_1 = d(F_0)$ (cf. Claim 4).

i. d, e must occur on the same process p, or else $G_1 = e(F_1) = e(d(F_0)) = d(G_0)$ will be 0-valent (cf. Lemma 2).

Intuition Whether the system decides on 0 or 1 entirely depends on the ordering of the two events d, e on process p. However, if p is slow, other processes will have to decide without p telling them about the ordering, which does not make sense.

ii. Consider all possible executions starting from F_0. By the requirement of termination, there must be an execution where:
   A. Some process decides, and
   B. Process p does not execute any steps (e.g., it fails).

   Let the state immediately after some process decides be T, where $T = \sigma(F_0)$ and σ does not contain any step by p.

iii. We have $e(T) = e(\sigma(F_0)) = \sigma(e(F_0)) = \sigma(G_0)$, which is 0-valent (cf. Lemma 2).

iv. We also have $e(d(T)) = e(d(\sigma(F_0))) = \sigma(e(d(F_0))) = \sigma(e(F_1)) = \sigma(G_1)$, which is 1-valent (cf. Lemma 2).

v. However, some process has already decided in T. Regardless of whether the decision is 0 or 1, agreement can be violated. This is a contradiction.

Proof of FLP theorem:

1. Start with some initial bivalent state (Lemma 1).
2. Assume processes take steps in round-robin fashion. Imaging that it is process p's turn.
3. (a) If the message system contains no messages for p, let $e = (p, \text{null})$.
   (b) Else, consider the oldest message m destined to p, and let $e = (p, m)$.
4. Let G be the current state.
   (a) If $e(G)$ is bivalent, execute (p, m).
   (b) Else, find a finite length schedule σ that does not contain e, and $e(\sigma(G))$ is bivalent (Lemma 4), and execute it in this order.
5. The system will always be in a bivalent state (i.e., no agreement).

Implications of FLP theorem: in practice, asynchronous distributed consensus is achievable with:

- Low probability of disagreement.
- Low probability of blocking/non-termination.
- Randomization.

8.5 *Version 4: node Byzantine failures (synchronous)*

Failure model:

- Nodes may experience byzantine failure (i.e., some nodes may behave maliciously).
- Communication channels are reliable.

Timing model: synchronous.

Under Byzantine failures, our goals are restricted to the non-faulty nodes:

- **Termination**: all non-faulty nodes eventually decide.
- **Agreement**: all non-faulty nodes should decide on the same value.
- **Validity**: if all non-faulty nodes have the same initial input, that value should be the only possible decision value. Else, nodes are allowed to decide on anything.

Theorem (*Byzantine consensus threshold*):

Let n be the total number of processes, and f be the number of possible Byzantine failures.

If $n \leq 3f$, then the Byzantine consensus problem cannot be solved.

We describe a protocol for $n \geq 4f + 1$. The underlying intuition is as follows:

- **Rotating coordinator paradigm**: processes are numbered from 1 to n, and process i is the coordinator for phase i.
- In each phase, the coordinator sends a proposal to all processes.
  - Each phase will need to have a *coordinator round* for this.
  - If the coordinator is non-faulty, all processes will see the proposal and arrive at a consensus.
  - If the coordinator is non-faulty for a phase, we say the phase is a **deciding phase**.
- To avoid a faulty coordinator from overruling the outcome of a deciding phase,
  - Processes do not listen to the coordinator if they see many identical values from other processes.
  - To achieve this, each phase will also have an *all-to-all broadcast round*.

Below is a formal description of the protocol:

```
1:  function RunProcess(i, input)
2:      int V[n];                                   ▷ initialized to all 0s
3:      V[i] = input;
4:      for k = 1 to f + 1 do
5:          send V[i] to all processes;             ▷ all-to-all broadcast round
6:          set V[1...n] to be the n values received;
7:          if value x occurs > n/2 times in V then
8:              proposal = x;
9:          else
10:             proposal = 0;
11:
12:         if k == i then                          ▷ coordinator round
13:             send proposal to all;               ▷ i is the coordinator
14:         receive coordinatorProposal from coordinator;
15:
16:         if value y occurs > n/2 + f times in V then   ▷ listen to coordinator?
17:             V[i] = y;
18:         else
19:             V[i] = coordinatorProposal;
20:         decide on V[i];
```

Proof of correctness:

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.

Proof This occurs due to lines 16 – 19.

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.

Case 1 Coordinator has proposal = x.

Proof On the coordinator, x must have appeared $> \frac{n}{2}$ times in V $\implies$ $> \frac{n}{2} - f$ must be from non-faulty processes.
On other processes, x appears $> \frac{n}{2} - f$ times in V $\implies$ impossible for another value y to appear $> \frac{n}{2} + f$ times in V.

cf. lines 7 – 8.

cf. line 16.

Case 2 Coordinator has proposal = 0.

Proof On the coordinator, no value x appears $> \frac{n}{2}$ times in V.
On other processes, impossible for y to appear $> \frac{n}{2} + f$ times in V.

cf. lines 9 – 10.

- **Termination**: obvious, since there are $f + 1$ phases.
- **Validity**: follows from Lemma 1.
- **Agreement**:

- With $f + 1$ phases, at least one of them is a *deciding phase*.
- Immediately after the deciding phase, all non-faulty processes P_i will have the same $V[i]$ (cf. Lemma 2).
- In the following phases, $V[i]$ on non-faulty processes does not change (cf. Lemma 1).

9 SELF-STABILIZATION

Lecture 10
5th April 2023

Motivation: distributed systems can get into illegal states due to faults, e.g.:

- Topology changes.
- Failures/Reboots.

Definitions:

- A **state** of a distributed system is the data state in all processes.
  - A state is either **legal** or **illegal**, defined based on application semantics.
  - The code on each process is assumed to be correct all the time.
- A distributed algorithm is **self-stabilizing** if:
  1. Starting from any (legal or illegal) state, the protocol all *eventually* reach a legal state *if there are no more faults*.
  2. Once the system is in a legal state, it will only transit to other legal states *unless there are faults*.

9.1 *Rotating privilege problem*

Given a ring of n processes, where each process can only communicate with its neighbours, "rotate" the **privilege** among all nodes.

- At any time, only one node may have the privilege.
- The node with the privilege may have exclusive access to some resource.
- Every node should have a fair chance at getting the privilege.

Algorithm:

- Each process i has a local integer variable $0 \leq V_i \leq k - 1$, where k is some constant no smaller than n.
- We single out one red process, and mark the other processes as blue.
- We assume that each process executes an action (defined below) atomically, i.e., an action does not interleave with other actions.

```
1:  function RedProcessAction
2:      let V be my value;
3:      retrieve value L of clockwise neighbor;
4:      if V == L then
5:          //complete whatever I want to do;
6:          V = (V + 1) % k;
7:
8:  function BlueProcessAction
9:      let V be my value;
10:     retrieve value L of clockwise neighbor;
11:     if V ≠ L then
12:         //complete whatever I want to do;
13:         V = L;
```

Notice that the algorithm is **self-stabilizing**, i.e., regardless of the initial values of the processes, the system will eventually get into a legal state and stay in legal states.

- In this problem, we define:
  - A process makes a **move** if (1) it has the privilege and (2) changes its value.
  - The system is in a **legal** state if exactly one process can make a move.
- Lemma: the following are legal states, and are the only legal states.
  1. All n values are the same.
  2. There are only two different values in the system, forming two consecutive bands, with one band starting from the red process.

  Proof These are legal states because there is exactly one process that can make a move in both cases. To prove that these are the only legal states, consider the value V of the red process and the value L of its clockwise neighbor.

  Case 1 V = L. Then, the red process can make a move, and no other process should be able to make a move (hence all n values must be the same).

  Case 2 V ≠ L. Then, the red process cannot make a move. Starting from it, find counterclockwise the first blue process whose value is different from its clockwise neighbor (such a blue process must exist since V ≠ L). This blue process can make a move, and no other process should be able to make a move (hence there are two bands).

Theorem (*legal → legal*):

If the system is in a legal state, then it will stay in legal states.

Proof Since actions are assumed to be executed atomically, we can define an ordering over all possible actions.

- For the red process, an action can only change the system state when V = L. When V = L, the only legal state is for all n values to be the same, hence updating V will result in two bands of values, which is also a legal state.
- For blue processes, an action can only change the system state when V ≠ L. When V ≠ L, the only legal state is to have two bands of values, and updating V to L will either result in all n values being the same or two bands of values. In either case, the system state is still legal.

Theorem (*illegal → legal*):

Regardless of the initial state of the system, it will eventually reach a legal state.

Lemma 1 Let P be a blue process, and Q be P's clockwise neighbor. If Q makes i moves, then P can make at most $i + 1$ moves.

Proof Based on the definition of a *move*, suppose the values of P and Q are initially different, P can make its first move and then set its value to Q's. Then, P will not be able to make any further moves until Q makes a move. Subsequently, for every move that Q makes, P can make 1 corresponding move.

Lemma 2 Let Q be the red process. If Q makes i moves, then there can be at most $ni + \frac{(n-1)n}{2}$ moves in the system.

Proof By repeatedly applying Lemma 1, we have $i + (i+1) + ... + (i+n-1) = ni + \frac{(n-1)n}{2}$.

Lemma 3 Consider a sequence of $\frac{n^2-n}{2} + 1$ moves in the system. The red process makes at least one move in the sequence.

Proof By contradiction. Assume that the red process makes 0 moves. Then, the entire system can make at most $\frac{(n-1)n}{2} = \frac{n^2-n}{2}$ moves (by Lemma 2), which is a contradiction.

Lemma 4 In any system state, there is always at least one process that can make a move.

Proof Consider the value V of the red process, and the value L of its clockwise neighbor. If V = L, the red process can make a move. If V ≠ L, there must exist some blue process whose value differs from its clockwise neighbor's, and that blue process can make a move.

Lemma 5 Regardless of the starting state, the system will eventually reach a state T where the red process has a different value from the values of any other process (though the system may not remain in such a state forever).

Note: as defined above, k is some constant no smaller than n.

Proof Let Q be the red process. If in the starting state, Q has the same value as some other process, then there must be an integer j, where $0 \leq j \leq k-1$ and j is not the value of any process. By Lemma 4, at least one process can make a move regardless of the state of the system, thus eventually the number of moves will approach infinity. From Lemma 3, Q will move at least once for every consecutive $\frac{n^2-n}{2}+1$ moves in the system. Thus, Q will eventually make an infinite number of moves, and eventually take j as its value.

* Note that before Q takes this value j, it is impossible for any blue process to have taken this value, because blue processes can only adopt the values from other process, and the value j was not present in the entire system.

Lemma 6 If the system is in a state where the red process has a different value from all other processes, it will eventually reach another state where all processes have the same value (though the system may not remain in such a state forever).

Proof Let the red process be P_1, and let P_1's value be x. Starting from the red process and counterclockwise along the ring, let the blue processes be $P_2, P_3, ..., P_n$.

* Let t be such that P_1 through P_t all have values x, and P_{t+1} through P_n all have values different from x.
* Initially, $t = 1$. We will prove that for any $1 \leq t \leq n-1$, t will increase by 1 at some point in time. Hence, eventually $t = n$, and we are done.
* To prove the above claim, note that P_{t+1} will take the value of x once it takes an action. Denote this event as E.
* Before event E, P_{t+2} through P_n can never take the value of x. Since P_n can never take the value of x before E, P_1 (and its subsequent nodes) can never change their values before E. Hence, immediately after E happens, $t \to t+1$.

Proof The proof for *illegal* $\to$ *legal* follows naturally from Lemmas 5 and 6.

9.2 *Self-stabilizing spanning tree*

Algorithm:

- Each process maintains two variables:
  - `parent`: details of my parent node (e.g., IP address + port number).
  - `dist`: my distance to root.
- The algorithm is executed periodically on all nodes in the background (i.e., `parent` and `dist` are continuously updated).
- We do not make the assumption that the algorithm is atomic.

- Note that at any given point in time, the values of the two variables can be wrong (due to faults such as topology change).

```
1: function RunProcess(id)
2:     if id = 1 then
3:         dist ← 0, parent ← null;
4:     else
5:         retrieve dist from all neighbours;
6:         dist ← 1 + smallest dist received;
7:         parent ← neighbour with smallest dist;     ▷ tie-break if necessary
```

Compared to the spanning tree covered in leader election (which is a one-shot algorithm), the self-stabilizing spanning tree will try to incrementally repair itself when something goes wrong.

Lecture 11
12th April 2023

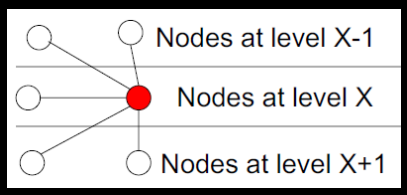
Before providing a correctness proof, we define:

- **Phase**: minimum time period where each process has executed its code at least once.
  - Some processes may end up taking multiple actions within a phase.
- **Level** of process i, A_i: length of the shortest path from process i to root.
- $dist_i$: value of `dist` (i.e., distance to root) on process i.
  - When the topology does not change (i.e., when there are no faults), A_i is fixed and $dist_i$ may change.

Also, notice that node levels have the following properties:

Property 1 A node at level X has at least one neighbor in level X − 1.

Property 2 A node at level X can only have neighbors in level X − 1, X, and X + 1.



Lemma 1 At the end of phase 1, $dist_1 = 0$ and $dist_i \geq 1$ for any $i \geq 2$.

Lemma 2 At the end of phase r,

- For any process i whose $A_i \leq r - 1$, we have $dist_i = A_i$.
- For any process i whose $A_i \geq r$, we have $dist_i \geq r$.

Proof By induction on r. We need to prove that at the end of phase $r + 1$,

- For any process i whose $A_i \leq r$, we have $dist_i = A_i$.
- For any process i whose $A_i \geq r + 1$, we have $dist_i \geq r + 1$.

Consider all t actions taken during phase $r + 1$.

Claim 1 The t actions will not rollback what is already achieved thus far by phase r.

Proof By induction on t. Base case is when $t = 0$, which is trivial. Next, assume that the statement holds for t, and consider action $t + 1$ by some node i.

Case 1 Let node i have $A_i \leq r - 1$. Thus it has at least one neighbor Y at level $A_i - 1$ (by Property 1). By our inductive hypothesis, we know that Y's `dist` value is $A_i - 1$.
Furthermore, node i only has neighbors at levels $A_i - 1$, A_i, and $A_i + 1$ (by Property 2). By our inductive hypothesis, the `dist` value on any of these neighbors will not be smaller than $A_i - 1$.
Hence, the min `dist` seen by node i will be $A_i - 1$, and node i will set its `dist` value to be A_i.

Case 2 Let node i have $A_i \geq r$. The proof is similar as above.

Claim 2 Each node i will satisfy the above conditions at some point in time during phase $r + 1$.

Proof By definition of a **phase**, node i takes at least one action in phase $r + 1$. Consider the following cases:

Case 1 Node i has $A_i \leq r - 1$. Since the `dist` values of its neighbors do not change throughout phase $r + 1$ (by Claim 1), the `dist` value of node i will not change as well (i.e., $dist_i = A_i$). Thus, it satisfies the condition.

Case 2 Node i has $A_i = r$. Then, it has at least one neighbor Y at level $A_i - 1$ (by Property 1), and the `dist` value of that node stays at $A_i - 1$ (by Case 1). Since node i only has neighbors at levels $A_i - 1$, A_i, and $A_i + 1$ (by Property 2), the min `dist` seen by node i will be $A_i - 1$. Thus, after node i takes an action, it will update its `dist` value to A_i, thus satisfying the condition.

Case 3 Node i has $A_i \geq r + 1$. For nodes with a neighbour at level A_i, and which take an action after that neighbour, their `dist` value will be set to $r + 1$. For nodes without such a neighbour, the `dist` values of their neighbours will remain $\geq r$ based on our induction hypothesis, thus their `dist` values after update will remain as $\geq r + 1$, satisfying the condition.

Claim 3 For each node, after it first satisfies the above conditions, it will continue to satisfy the conditions for the remainder of phase $r + 1$.

Proof Enumerate the 3 cases (similar to in Claim 2), and leverage the that the conditions always hold throughout phase $r + 1$ (by Claim 1).

Based on the above claims, we have shown that the above conditions hold at the end of phase $r + 1$.

Theorem After $H + 1$ phases, $dist_i = A_i$ on all processes.

– H rep. the eccentricity of the graph (i.e., length of the shortest path from the process furthest away from root, to the root).

Proof Directly follows from the earlier lemma.

Theorem After $H + 1$ phases, the `dist` and `parent` values on all processes are correct.

Proof Each process has a single parent pointer except the root. Thus, the graph has n nodes and $n - 1$ edges. Each process has a path to the root, thus the graph is connected. Hence, it is a spanning tree.

9.3 *Remark: correctness proofs for self-stabilization*

The following proof technique is typical for proving self-stabilization:

Consider all t actions taken during phase $r + 1$.

1. **No backward move**: prove that the t actions will not rollback what is already achieved thus far by phase r.
2. **Forward move**: prove that at some point, each node will achieve more.
3. **No backward move after forward move**: prove that the t actions will not roll back the effects of the forward move after that forward move happens.

* * *