CS4231 Parallel and Distributed Algorithms

Synchronisation

- **Critical section**
  - Mutual exclusion (no more than one process in CS)
  - Progress (if one or more processes wants to enter and no one is in CS, then someone can enter)
  - No starvation (if a process wants to enter, it can eventually always enter, even in worst-case schedule)

- **Peterson's algorithm**

```
boolean wantCS[i] = {false};
int turn = 0;
```

```
// Process 0
RequestCS(0) {
   wantCS[0] = true;
   turn = 1;
   while (wantCS[1] == true
      && turn == 1);
}
ReleaseCS(0) {
   wantCS[0] = false;
}
```

```
// Process 1
RequestCS(1) {
   wantCS[1] = true;
   turn = 0;
   while (wantCS[0] == true
      && turn == 0);
}
ReleaseCS(1) {
   wantCS[1] = false;
}
```

- **Lamport's bakery algorithm**
  - When a process arrives, get a queue number larger than everyone else
  - Due to interleaving, two processes might get the same queue number; then be break ties by id

```
boolean choosing[i] = {false};
int number[i] = {0};
ReleaseCS(int myid) {
   number[myid] = 0;
}
boolean Smaller(int number1, int id1, int number2, int
    id2) {
   if (number1 < number 2) return true;
   if (number1 == number2) {
      if (id1 < id2) return true; else return false;
   }
   if (number 1 > number2) return false;
}
RequestCS(int myid) {
   choosing[myid] = true;
   for (int j = 0; j < n; j++)
      if (number[j] > number[myid]) number[myid] =
          number[j];
   number[myid]++;
   choosing[myid] = false;

   for (int j = 0; j < n; j++) {
      while (choosing[j] == true);
      while (number[j] != 0 &&
                Smaller(number[j], j, number[myid],
                   myid));
   }
}
```

- **Dekker's algorithm**

```
boolean wantCS[] = {false, false};
int turn = 1;
public void requestCS(int i) { // entry protocol
   int j = 1 - i;
   wantCS[i] = true;
   while (wantCS[j]) {
      if (turn == j) {
         wantCS[i] = false;
         while (turn == j); // busy wait
         wantCS[i] = true;
      }
   }
}
public void releaseCS(int i) { // exit protocol
   turn = 1 - i;
   wantCS[i] = false;
}
```

- **Hardware solutions**
  - Disabling interrupts to prevent context switch
  - Special machine-level instructions (TestAndSet)

- **Semaphore**
  - no busy waiting

```
value = true;
queue = {};
P() {
   if (value == false) {
      add myself to queue and block;
   }
   value = false;
}
V() {
   value = true;
   if (queue is not empty) {
      wake up one arbitrary process on the queue;
   }
}
```

  - exactly one process is waken up in V(), usually chosen arbitrary (some implementations always choose the first in queue)
  - can implement CS using semaphore

- **Dining philosophers problem**
  - Solution: pick up the lower-indexed chopstick first, to avoid a cycle

- **Monitor**
  - every Java object is a monitor
  - two queues: normal CS queue, and CV (wait/notify/notifyAll) queue
  - Hoare-style monitor: `object.notify()` immediately transfers control to awakened thread
  - Java-style monitor: `object.notify()` awakened thread still needs to queue with other threads (no specific priority) (i.e. thread transits from "Waiting" to "Blocked" state)

- **Single-producer single-consumer with monitors**

```
void produce() {
   synchronized
      (sharedBuf) {
      while (sharedBuf is
         full)
         sharedBuf.wait();
      add an item to
         buffer;
      if (buffer *was*
         empty)
         sharedBuf.notify();
   }
}
```

```
void consume() {
   synchronized
      (sharedBuf) {
      while (sharedBuf is
         empty)
         sharedBuf.wait();
      remove an item from
         buffer;
      if (buffer *was*
         full)
         sharedBuf.notify();
   }
}
```

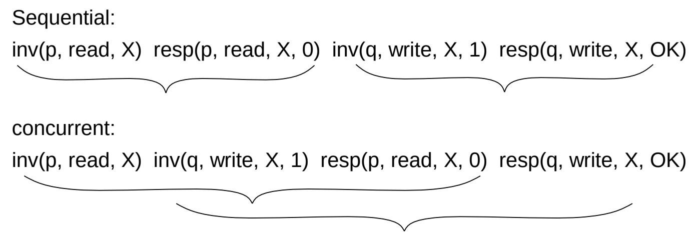
- **(Multiple) reader-writer with monitors** - writers might get starved if there is a continuous stream of readers

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

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

Consistency

- **Sequential history / concurrent history**
  - A history is sequential if any invocation is always *immediately* followed by its response
  - Otherwise it is concurrent


Sequential:

$\underbrace{\text{inv(p, read, X)}\ \ \text{resp(p, read, X, 0)}}\ \ \underbrace{\text{inv(q, write, X, 1)}\ \ \text{resp(q, write, X, OK)}}$

concurrent:

inv(p, read, X) inv(q, write, X, 1) resp(p, read, X, 0) resp(q, write, X, OK)

- ***Legal* sequential history**
  - Sequential history that satisfies *sequential semantics* of the data type (i.e. 'read' operations should produce expected results)

- **Equivalence**
  - Two histories are equivalent if they have exactly the same set of events (need not be in the same order)

- **Process order**
  - Event a is before event b if they are on the same process and they a is executed before b

- **Sequential consistency**
  - A history is sequentially consistent if it is equivalent to some legal sequential history that preserves process order

- **Subhistory ($H \mid p$) of a process, or ($H \mid o$) of an object**
  - Subsequence of all events in p, or all objects of o

- **External order (*occurs-before*)**
  - Partial order, $o1 < o2$ iff response of $o1$ appears before invocation of $o2$

- **Linearizability (intuitive definition)**
  - Execution is equivalent to some execution where each operation happens instantaneously at some point between invocation and response

- **Linearizability (formal definition)**
  - History H is equivalent to some legal sequential history S, and the external order induced by H is a subset of the external order induced by S

- **Linearizable $\implies$ sequentially consistent**

- **Linearizability is a *local property***
  - H is linearizable $\iff$ $\forall$ object x, $H \mid x$ is linearizable

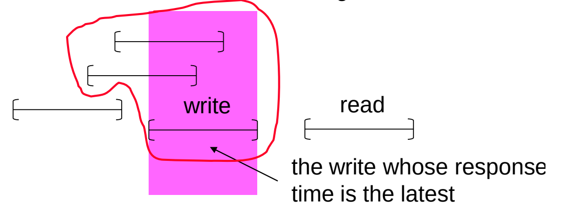
- **Sequential consistency is not a *local property***

Consistency Definitions for Registers

- **Atomic $\iff$ linearizable**

- **Regular**
  - 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
  - Definition of most recent writes

An atomic register must be regular



- **Safe**
  - When a read does not overlap with any write, then it returns the value written by one of the most recent writes (same as Regular)
  - When a read overlaps with one or more writes, it can return anything

- **Atomic $\implies$ regular $\implies$ safe**

- **Regular $\not\Leftrightarrow$ sequential consistent**

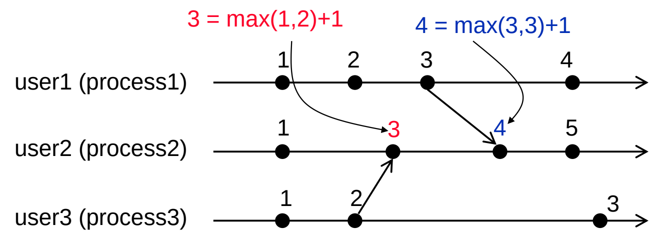
Models and Clocks

- **Software clock**
  - No relation to physical time
  - Allows a protocol to infer ordering among events

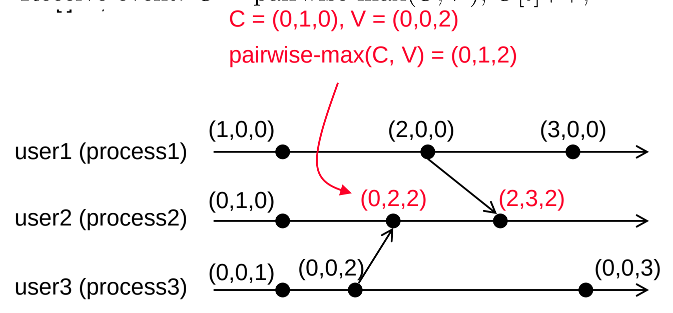
- ***Happens-before***
  - partial order that considers process order, send-receive order, transitivity

- **Logical clock**
  - Increment local counter at every "local computation" and "send" event

- Receive event: $C = \max(C, V) + 1$

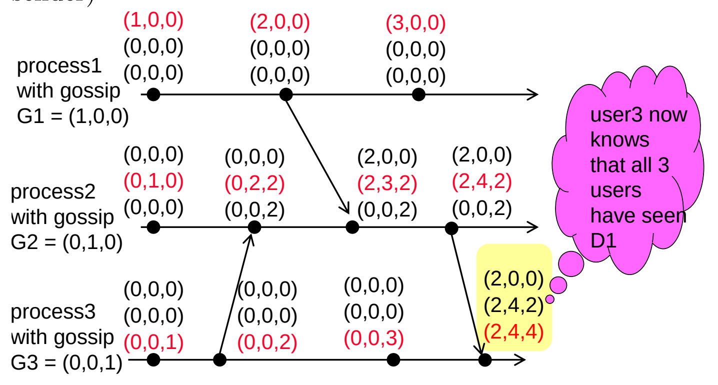


- **Vector clock**
  - Increment $C[i]$ of local vector at every "local computation" and "send" event
  - Receive event: C = pairwise-max(C, V); $C[i]$++;



- Logical clock:
  s happens before $t \implies$ logical-clock$(s) <$ logical-clock(t)
  Vector clock:
  s happens before $t \iff$ logical-clock$(s) <$ logical-clock(t)
  where "$<$" := every field in $v1$ is less than or equal to the corresponding field in $v2$, and at least one field is *strictly* less

- **Matrix clock**
  - Overview:
  — Each process maintains n vector clocks, one containing data from each process
  — i^{th} vector on process i is called process i's principle vector
  — Principle vector is the same as vector clock
  — Non-principle vectors are just piggybacked on messages to update "knowledge"
  - Increment $C[i]$ (C := principle vector of process i) at every "local computation" and "send" event
  - Receive event (principle vector C):
  C = pairwise-max(C, V); $C[i]$++; (V := principle vector of sender)
  - Receive event (non-principle vector C):
  C = pairwise-max(C, V); (V := corresponding vector of sender)



- Vector clock: Know what I have seen
  Matrix clock: Know what other people have seen

Snapshots

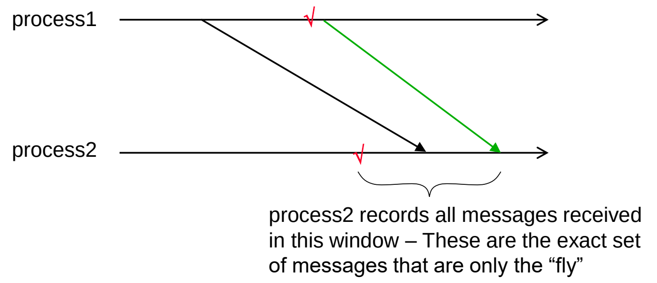
- **Global snapshot**
  - A set of events such that if $e2$ is in the set and $e1$ is before $e2$ in process order, then $e1$ must be in the set

- ***Consistent* global snapshot**
  - A 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
  - i.e. Consistent global snapshot captures all *happens-before* relationships

- For any event e_i of any process, there exists a consistent global snapshot S where:

$$\begin{cases} e_i \in S & \text{if } i \le m \\ e_i \notin S & \text{if } i > m \end{cases}$$

- **Protocol for consistent global snapshot** (assume FIFO delivery on each channel)
  - Initiated by one process
  - After each process takes a snapshot, it sends out a control message to all other processes
  - If a process receives a control message but has not taken a snapshot, it takes a snapshot immediately
  - For each pair of processes s and r, the messages received between r's snapshot time and the control message from s to r are considered to be *on-the-fly*, and they are recorded by r upon receipt


process1
process2
process2 records all messages received in this window – These are the exact set of messages that are only the "fly"

Causal order / Total order

- **Causal order**
  - If $s1$ *happened before* $s2$, and $r1$ and $r2$ are on the same process, then $r1$ must be before $r2$

- **Protocol for ensuring casual order**
  - Each process maintains an n by n matrix M
  - $M[i, j]$:= num. of messages sent from i to j
  - Before i sends a message to j, do $M[i, j]$++ before piggybacking M with the message
  - Deliver the message and set local matrix M = pairwise-max(M, T) if:

$$\begin{cases} T[k, j] \le M[k, j] & \forall k \ne i \\ T[i, j] = M[i, j] + 1 \end{cases}$$

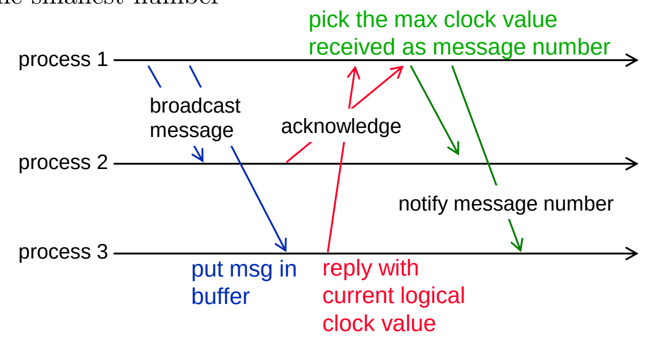
  - Otherwise delay message

- **Total order (when broadcasting messages)**
  - All messages delivered to all processes in exactly the same order

- Total order $\not\Leftrightarrow$ casual order

- Total order broadcast protocol using a designated coordinator

- **Skeen's algorithm for Total Order Broadcast**
  - Each process maintains logical clock and a message buffer for undelivered messages
  - A message in the buffer is delivered if: all messages in the buffer have been assigned numbers, and this message has the smallest number

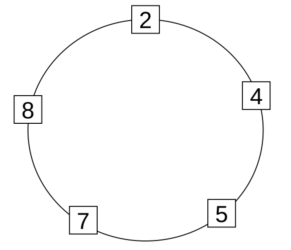


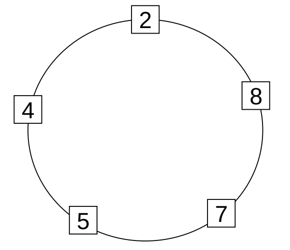
Leader Election

- **Leader election on *anonymous* ring**
  - In the sense that there are no unique identifiers for each process
  - It is impossible using deterministic algorithms (by symmetry)
  - For unknown ring size it is not possible to terminate (i.e. to be certain that there is a unique leader after a finite number of steps)
  Randomized algorithm that terminates with probability 1 on known ring size:
  - At each phase, run Chang-Roberts algorithm (with hop count)
  - If a node receives its own message, then it is one of the winners, so it proceeds to the next phases
  - Losers only forward messages in future phases
  - If there is only a single winner (can be detected by winning mode), then the algorithm stops

- **Leader election on ring: Chang-Roberts algorithm**
  - Each node has a unique id
  - Nodes send election message with its own id clockwise
  - Election message is forwarded if id in message is larger than own id
  - Otherwise message is discarded
  - (When a node receives its own election message, it becomes the leader)

- **Chang-Roberts algorithm message complexity**
  - Best case:
    - 2n-1 messages
  - Worst case:
    - n(n-1)/2 messages



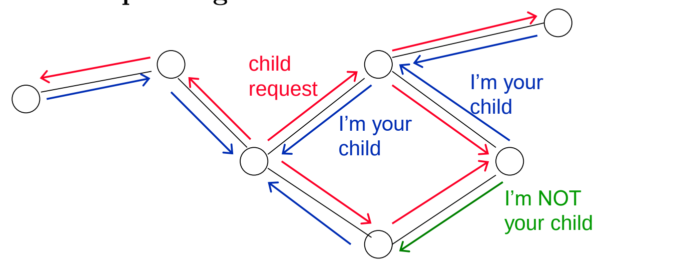


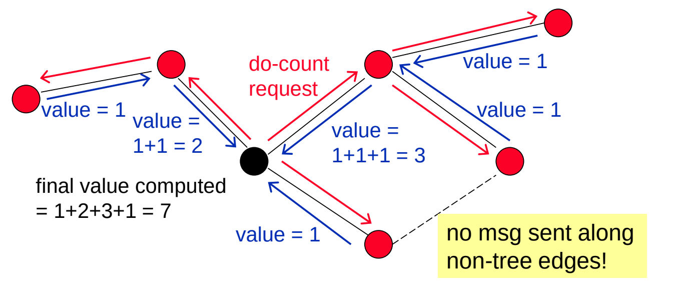
  - Average case can be proved to be $O(n \log n)$.

- **Leader election on complete graph**
  - Each node sends its id to all other nodes
  - Wait until you receive n ids
  - Biggest id wins

- **Leader election on any connected graph**
  - Flood your id to all other nodes (using spanning tree edges)
  - Wait until you receive n ids
  - Biggest id wins

- **Spanning tree construction / Count number of nodes on spanning tree**





- **Spanning tree usages**
  - Broadcast
  - Any aggregation (count, sum, average, max, min, etc.)

Agreement / Consensus

- **Crash failure**: Node stops sending any more messages after crashing

- **Byzantine failure**: Node can send arbitrary messages after failing

- **Synchronous**: Message delay has a known upper bound x, and node processing delay has a known upper bound y (so that it is possible to accurately detect crash failures)
  - For synchronous timing model, processes can always proceed in inter-locked rounds, where in each round:
  — Every process sends one message to every other process
  — Every process receives one message from every other process
  — Every process does some local computation
  (Can be implemented with accurate clocks, or clocks with bounded error)

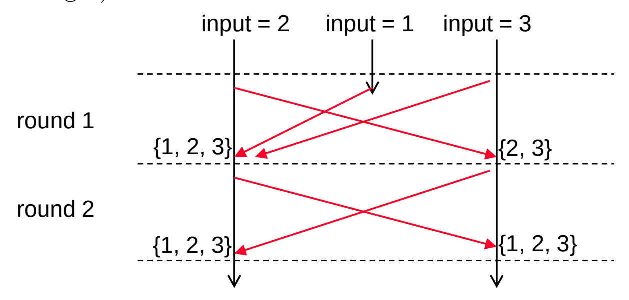
- **Asynchronous**: Message delay is finite but unbounded

- **Goals**
  Termination: All non-failed nodes eventually decide
  Agreement: All non-failed nodes should decide on the same value
  Validity: If all nodes have the same initial input, then that value must be the decision value

- **Ver0: No node or link failures**: Trivial

- **Ver1: Node crash failures, channels are reliable, synchronous**
  - Forward messages to all other nodes for $(f + 1)$ rounds to tolerate f failures
  (Key idea for proof: If there is a round in which no node fails, then every non-failed node will have the same set of

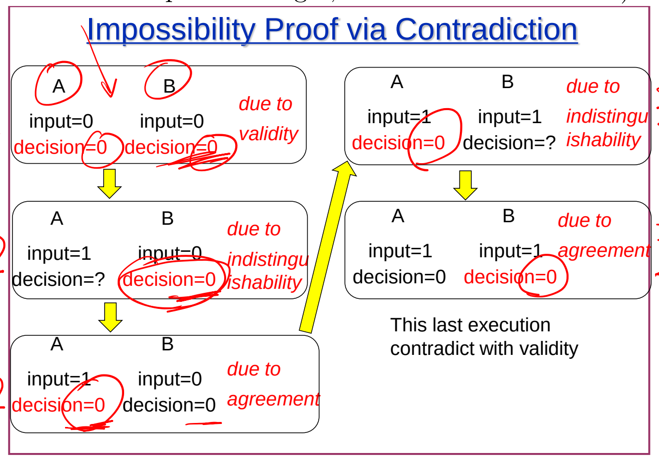
messages)



```
S <- {my input}
for (int i = 1; i <= f+1; i++) {
   // do this for f+1 rounds
   send S to all other nodes
   receive n-1 sets;
   for each set T received: S <- Union(S, T)
}
Decide on min(S);
```

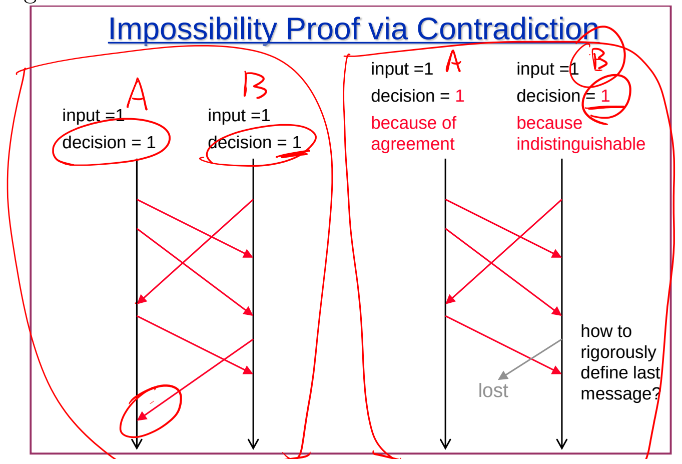
- It can be shown that *any* consensus protocol will take at least $(f+1)$ rounds

- **Ver2: No node failures, channels may drop messages, synchronous**
  - It is immpossible to reach goal using a deterministic algorithm (can consider the case where the communication channel can drop all messages, do we decide on 0 or 1?)

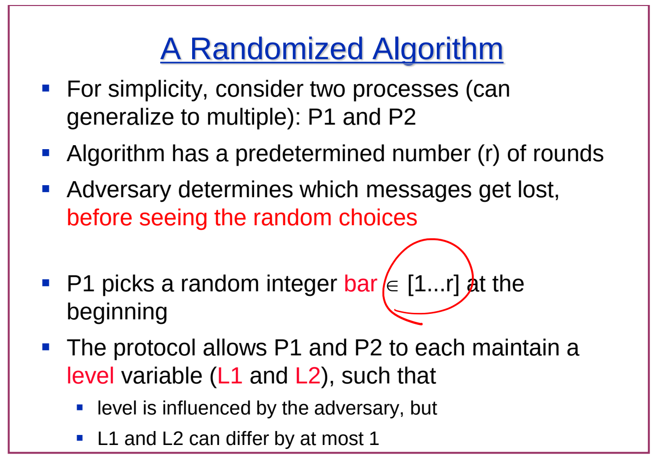

Impossibility Proof via Contradiction

This last execution contradict with validity

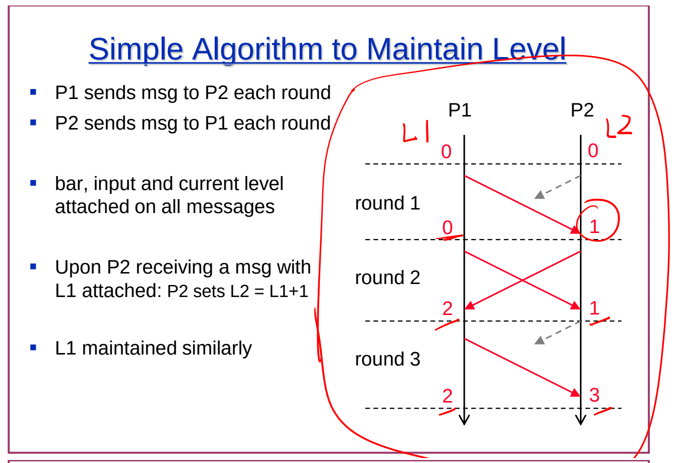
- Even with weakened validity requirement (decision is required to be 1 only if no message is lost throughout the execution), it is still impossible using a deterministic algorithm


Impossibility Proof via Contradiction

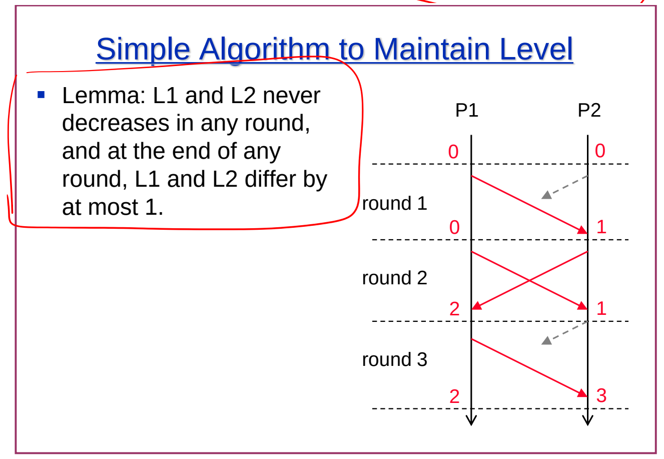
- With *limited disagreement* (all nodes must agree with probability $(1-p_{error})$) and weakened validity, it is possible


A Randomized Algorithm

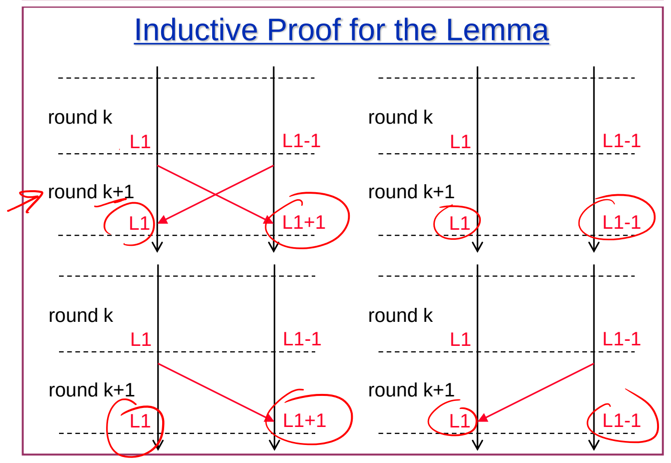
- For simplicity, consider two processes (can generalize to multiple): P1 and P2
- Algorithm has a predetermined number (r) of rounds
- Adversary determines which messages get lost, before seeing the random choices
- P1 picks a random integer bar ∈ [1...r] at the beginning
- The protocol allows P1 and P2 to each maintain a level variable (L1 and L2), such that
  - level is influenced by the adversary, but
  - L1 and L2 can differ by at most 1

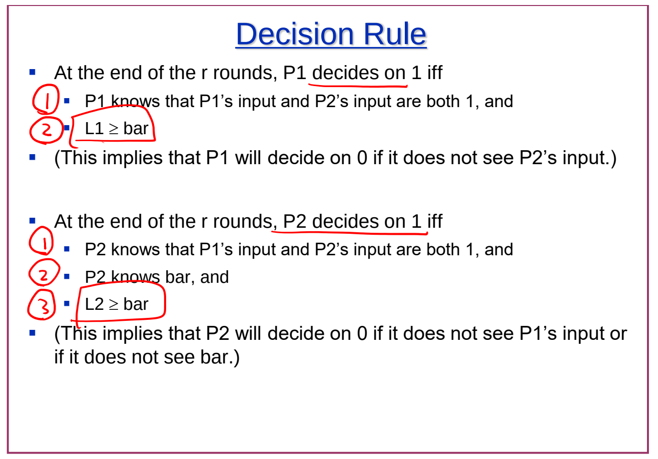

Simple Algorithm to Maintain Level

- P1 sends msg to P2 each round
- P2 sends msg to P1 each round
- bar, input and current level attached on all messages
- Upon P2 receiving a msg with L1 attached: P2 sets L2 = L1+1
- L1 maintained similarly

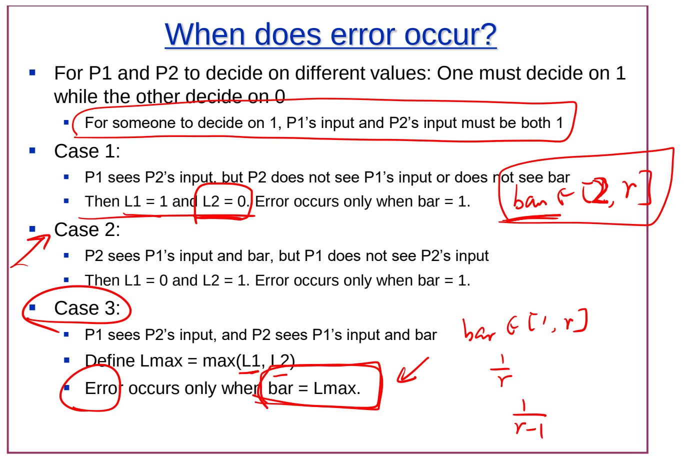

Simple Algorithm to Maintain Level

- Lemma: L1 and L2 never decreases in any round, and at the end of any round, L1 and L2 differ by at most 1.


Inductive Proof for the Lemma


Decision Rule

- At the end of the r rounds, P1 decides on 1 iff
  1. P1 knows that P1's input and P2's input are both 1, and
  2. L1 ≥ bar
- (This implies that P1 will decide on 0 if it does not see P2's input.)
- At the end of the r rounds, P2 decides on 1 iff
  1. P2 knows that P1's input and P2's input are both 1, and
  2. P2 knows bar, and
  3. L2 ≥ bar
- (This implies that P2 will decide on 0 if it does not see P1's input or if it does not see bar.)


When does error occur?

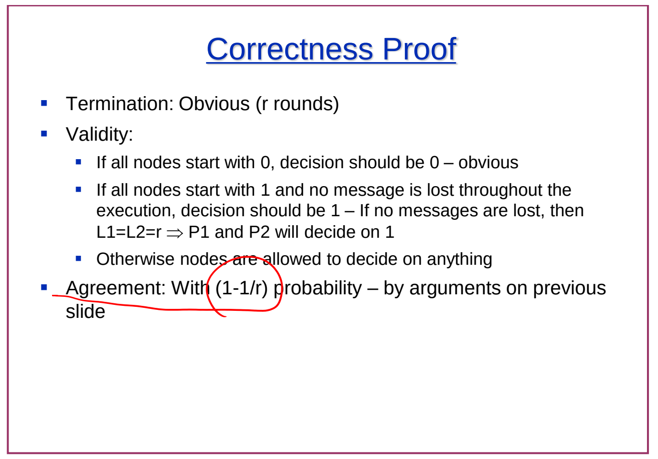
- For P1 and P2 to decide on different values: One must decide on 1 while the other decide on 0
  - For someone to decide on 1, P1's input and P2's input must be both 1
- Case 1:
  - P1 sees P2's input, but P2 does not see P1's input or does not see bar
  - Then L1 = 1 and L2 = 0. Error occurs only when bar = 1.
- Case 2:
  - P2 sees P1's input and bar, but P1 does not see P2's input
  - Then L1 = 0 and L2 = 1. Error occurs only when bar = 1.
- Case 3:
  - P1 sees P2's input, and P2 sees P1's input and bar
  - Define Lmax = max(L1, L2)
  - Error occurs only when bar = Lmax.

bar ∈ [2, r]

bar ∈ [1, r]

$\frac{1}{r}$

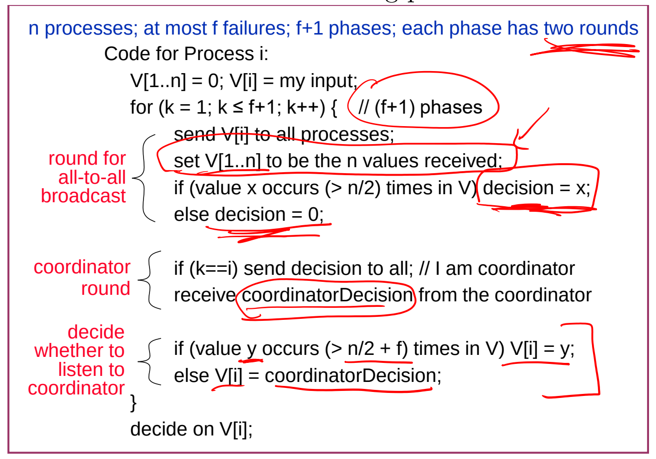
$\frac{1}{r-1}$


Correctness Proof

- Termination: Obvious (r rounds)
- Validity:
  - If all nodes start with 0, decision should be 0 – obvious
  - If all nodes start with 1 and no message is lost throughout the execution, decision should be 1 – If no messages are lost, then L1=L2=r ⇒ P1 and P2 will decide on 1
  - Otherwise nodes are allowed to decide on anything
- Agreement: With (1-1/r) probability – by arguments on previous slide

- **Ver3: Node crash failures, channels are reliable, asynchronous**
  - FLP theorem states that this is impossible to solve even with:
  — up to one single node crash failure
  — FIFO ordering within each channel
  — non-blocking receive

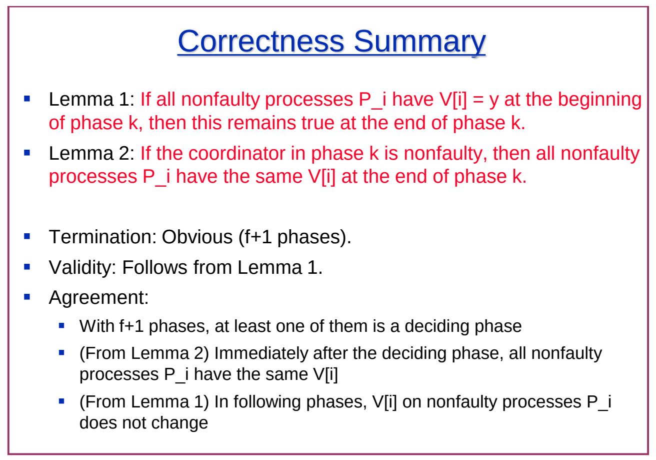
- **Ver4: Node Byzantine failures, channels are reliable, synchronous**
  - n := total number of processes
  - f := number of possible Byzantine failures
  - Can be shown to be impossible to solve if $n \leq 3f$
  Protocol for $n \geq 4f+1$:
  - High-level idea:
  — Run $(f+1)$ phases (each phase contains a fixed number of rounds)
  — Each phase has a *coordinator* node
  — A phase is a *deciding* phase if the coordinator is non-faulty
  — There must be at least one deciding phase, and after the deciding phase it is impossible for a subsequent faulty coordinator to overrule the deciding phase


n processes; at most f failures; f+1 phases; each phase has two rounds

```
Code for Process i:
    V[1..n] = 0; V[i] = my input;
    for (k = 1; k ≤ f+1; k++) {   // (f+1) phases
        send V[i] to all processes;
        set V[1..n] to be the n values received;
        if (value x occurs (> n/2) times in V) decision = x;
        else decision = 0;

        if (k==i) send decision to all; // I am coordinator
        receive coordinatorDecision from the coordinator

        if (value y occurs (> n/2 + f) times in V) V[i] = y;
        else V[i] = coordinatorDecision;
    }
    decide on V[i];
```
round for all-to-all broadcast
coordinator round
decide whether to listen to coordinator


Correctness Summary

- Lemma 1: If all nonfaulty processes P_i have V[i] = y at the beginning of phase k, then this remains true at the end of phase k.
- Lemma 2: If the coordinator in phase k is nonfaulty, then all nonfaulty processes P_i have the same V[i] at the end of phase k.
- Termination: Obvious (f+1 phases).
- Validity: Follows from Lemma 1.
- Agreement:
  - With f+1 phases, at least one of them is a deciding phase
  - (From Lemma 2) Immediately after the deciding phase, all nonfaulty processes P_i have the same V[i]
  - (From Lemma 1) In following phases, V[i] on nonfaulty processes P_i does not change

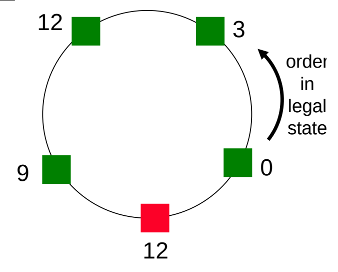
Self-Stabilization

- **Legal state**
  - Defined by application semantics

- **Self-stabilizing system**: System where:
  - Starting from any state, the protocol will eventually reach a legal state if there are no more faults
  - Once the system is in a legal state, it will only transit to other legal states unless there are faults

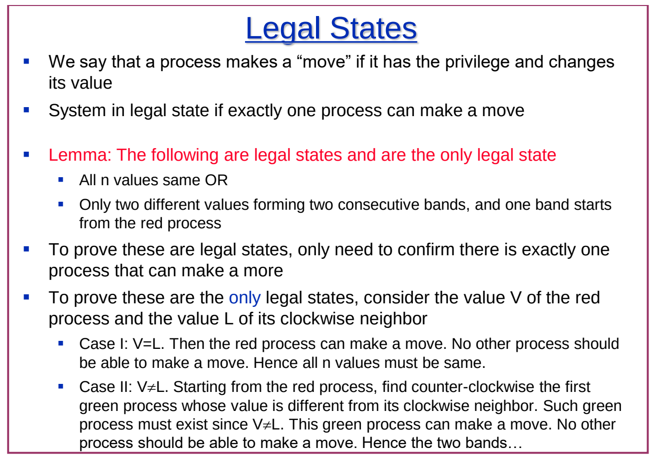
- **Rotating privilege algorithm**
  - Each process i has a local integer variable V_i, where $0 \leq V_i < k$ for some constant $k \geq n$

Example: n = 5 and k = 13



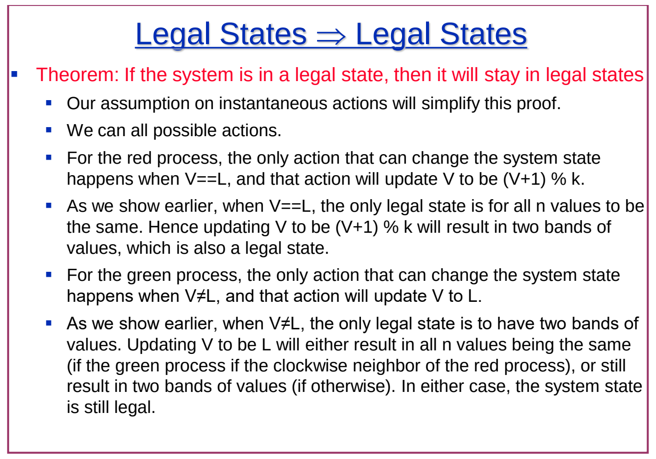
```
// Red process
L <- value of clockwise
     neighbor;
V <- my value;
if (V == L) {
   // do privileged action
   V = (V+1) % k;
}
```

```
// Green process
L <- value of clockwise
     neighbor;
V <- my value;
if (V != L) {
   // do privileged action
   V = L;
}
```



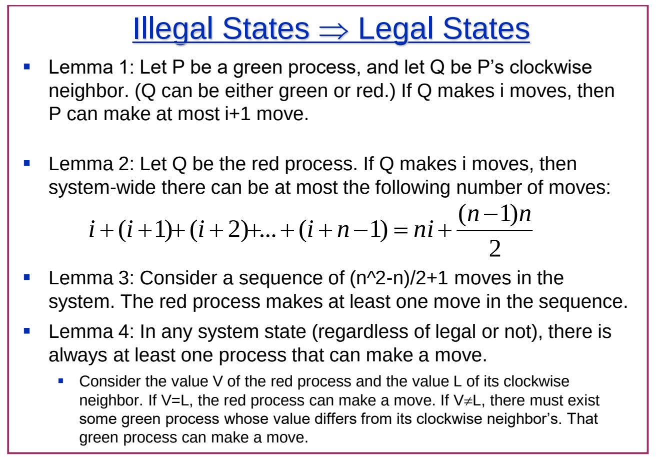
Legal States

- We say that a process makes a "move" if it has the privilege and changes its value
- System in legal state if exactly one process can make a move
- Lemma: The following are legal states and are the only legal state
  - All n values same OR
  - Only two different values forming two consecutive bands, and one band starts from the red process
- To prove these are legal states, only need to confirm there is exactly one process that can make a more
- To prove these are the only legal states, consider the value V of the red process and the value L of its clockwise neighbor
  - Case I: V=L. Then the red process can make a move. No other process should be able to make a move. Hence all n values must be same.
  - Case II: V≠L. Starting from the red process, find counter-clockwise the first green process whose value is different from its clockwise neighbor. Such green process must exist since V≠L. This green process can make a move. No other process should be able to make a move. Hence the two bands…



Legal States ⇒ Legal States

- Theorem: If the system is in a legal state, then it will stay in legal states
  - Our assumption on instantaneous actions will simplify this proof.
  - We can all possible actions.
  - For the red process, the only action that can change the system state happens when V==L, and that action will update V to be (V+1) % k.
  - As we show earlier, when V==L, the only legal state is for all n values to be the same. Hence updating V to be (V+1) % k will result in two bands of values, which is also a legal state.
  - For the green process, the only action that can change the system state happens when V≠L, and that action will update V to L.
  - As we show earlier, when V≠L, the only legal state is to have two bands of values. Updating V to be L will either result in all n values being the same (if the green process if the clockwise neighbor of the red process), or still result in two bands of values (if otherwise). In either case, the system state is still legal.

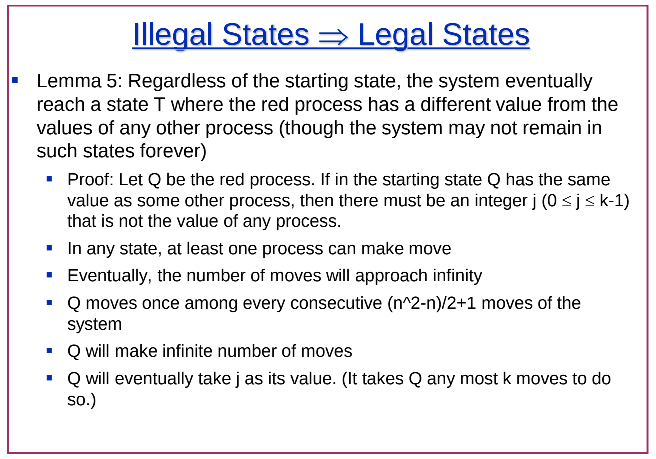


Illegal States ⇒ Legal States

- Lemma 1: Let P be a green process, and let Q be P's clockwise neighbor. (Q can be either green or red.) If Q makes i moves, then P can make at most i+1 move.
- Lemma 2: Let Q be the red process. If Q makes i moves, then system-wide there can be at most the following number of moves:

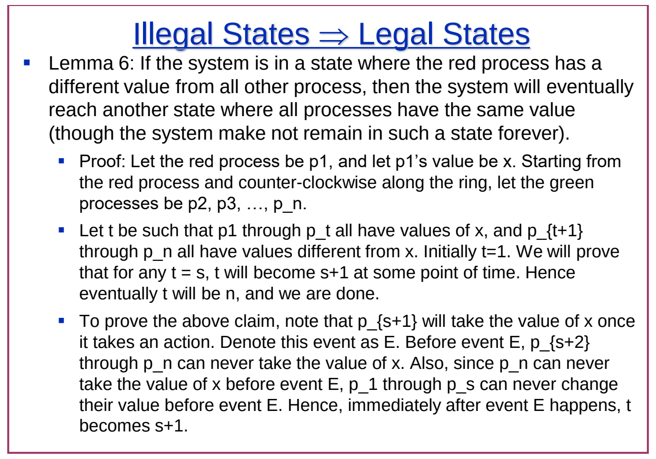
$$i+(i+1)+(i+2)+\ldots+(i+n-1)=ni+\frac{(n-1)n}{2}$$

- Lemma 3: Consider a sequence of (n^2-n)/2+1 moves in the system. The red process makes at least one move in the sequence.
- Lemma 4: In any system state (regardless of legal or not), there is always at least one process that can make a move.
  - 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 green process whose value differs from its clockwise neighbor's. That green process can make a move.



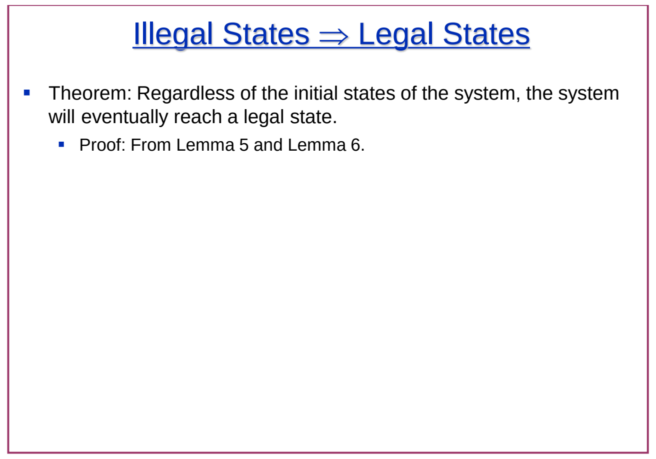
Illegal States ⇒ Legal States

- Lemma 5: Regardless of the starting state, the system 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 states forever)
  - 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 (0 ≤ j ≤ k-1) that is not the value of any process.
  - In any state, at least one process can make move
  - Eventually, the number of moves will approach infinity
  - Q moves once among every consecutive (n^2-n)/2+1 moves of the system
  - Q will make infinite number of moves
  - Q will eventually take j as its value. (It takes Q any most k moves to do so.)



Illegal States ⇒ Legal States

- Lemma 6: If the system is in a state where the red process has a different value from all other process, then the system will eventually reach another state where all processes have the same value (though the system make not remain in such a state forever).
  - Proof: Let the red process be p1, and let p1's value be x. Starting from the red process and counter-clockwise along the ring, let the green processes be p2, p3, …, p_n.
  - Let t be such that p1 through p_t all have values of x, and p_{t+1} through p_n all have values different from x. Initially t=1. We will prove that for any t = s, t will become s+1 at some point of time. Hence eventually t will be n, and we are done.
  - To prove the above claim, note that p_{s+1} will take the value of x once it takes an action. Denote this event as E. Before event E, p_{s+2} through p_n can never take the value of x. Also, since p_n can never take the value of x before event E, p_1 through p_s can never change their value before event E. Hence, immediately after event E happens, t becomes s+1.



Illegal States ⇒ Legal States

- Theorem: Regardless of the initial states of the system, the system will eventually reach a legal state.
  - Proof: From Lemma 5 and Lemma 6.

- **Self-stabilizing spanning tree algorithm**
  - On the root node: `dist = 0; parent = null;`
  - On other nodes (executed periodically):
  — Retrieve `dist` from all neighbours
  — Set my own `dist = 1 + <smallest dist received>` and `parent = <neighbour with smallest dist>` (tie-break if necessary)

  Proof:
  - Define a *phase* to be the minimum time period where each process has taken an action
  - $A_i :=$ length of the real shortest path from node i to root
  - Lemma 1: At the end of phase 1, $dist_{root} = 0$ and $dist_i \geq 1$ for all other i
  - Lemma 2: At the end of phase r:
  — Any process i with $A_i < r$, we have $dist_i = A_i$
  — Any process i with $A_i \geq r$, we have $dist_i \geq r$
  - Prove by induction using Lemma 1 and Lemma 2