CS4231 2324 Sem 2 Final Cheatsheet, by Stanve Widjaja. (Main ref: Bernard)

1 Synchronisation

Critical Section. Properties needed: {Mutex (mutual exclusion): no more than 1 proc in CS; Progress: if one or more proc wants to enter AND if no one in CS, then 1 of them can eventually enter CS (intuitively, w.t. ensure that "resource" is fully utilized); No starvation: if a proc wants to enter, it eventually can always enter (no starvation implies progress)}.

Always need to consider the WORST-CASE schedule. If a proc is in CS, we ALWAYS assume that it will eventually exit the CS.

- **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

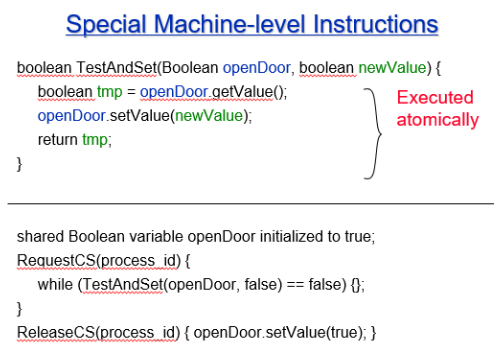
```
boolean choosing[i] = {false};
int number[i] = {0};
ReleaseCS(int myid) {
   number[myid] = 0;
}
boolean Smaller(int number1, int id1, int number2, int
     id2) {
   if (number1 < number 2) return true;
   if (number1 == number2) {
      if (id1 < id2) return true; else return false;
   }
   if (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 Algo:

```
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. They are {1. Disabling interrupts to prevent context switch; 2. Special machine-level instructions}.



Special Machine-level Instructions

```
boolean TestAndSet(Boolean openDoor, boolean newValue) {
   boolean tmp = openDoor.getValue();
   openDoor.setValue(newValue);
   return tmp;
}
```

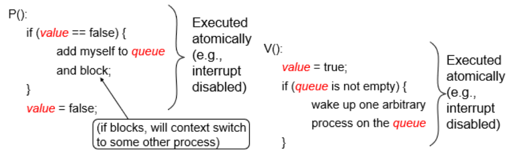
Executed atomically

```
shared Boolean variable openDoor initialized to true;
RequestCS(process_id) {
   while (TestAndSet(openDoor, false) == false) {};
}
ReleaseCS(process_id) { openDoor.setValue(true); }
```

2 Semaphores, Monitors

Busy Wait Problem: wastes CPU cycles (want to release CPU to other proc-s); need OS support. In this lecture, we discuss synchro primitives (OS-level APIs that the program may call — no need to worry abt how they are imple-d).

Semaphore. No busy waiting. Internally, each sema has boolean *value*, initially true + *queue* of blocked proc-s, initially empty. Both these internal functions are executed atomically (e.g. interrupt disabled).



```
P():
   if (value == false) {
         add myself to queue
         and block;
   }
   value = false;
```

Executed atomically (e.g., interrupt disabled)

(if blocks, will context switch to some other process)

```
V():
   value = true;
   if (queue is not empty) {
         wake up one arbitrary
         process on the queue
   }
```

Executed atomically (e.g., interrupt disabled)

Imple CS using semaphore: request "just call $P()$", release "just call $V()$".

Dining Philo-s Problem. Solution: pick up the lower-indexed chopstick first, to avoid a cycle.

Monitor. Basic Primitives: {Every Java object is (i.e. can be made into) a monitor; two queues: {normal CS queue and an additional [wait/notify/notifyAll] queue}}.

2 kinds of monitors: {Hoare-style monitor: *obj.notify*() immediately transfers control to awakened thread; Java-style monitor: *obj.notify*() awakened thread still needs to queue with other threads (no specific priority) (i.e. thread transits from "waiting" to "blocked" state)}.

More Problems. {1-producer 1-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 (note: 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();
   }
}
```

}

3 Consistency

i.e. the lecture notes among 1-7 with the most notations and formalization.

ADT. A piece of data with allowed operations on the data: {Integer X: {$read()$, $write()$, $increment()$}; Queue: {$enqueue()$, $dequeue()$}; Stack: {$push()$, $pop()$}}. We consider shared ADT (can be accessed by multiple proc-s, shared obj as a shorthand).

Consistency? A term with a thousand defn-s.

Defn in this course: {Consistency specifies what behavior is allowed WHEN a shr-d obj is accessed by multiple proc-s; when we say sth is "consistent", we mean it satisfies the spec (according to some given/particular spec)}. Spec(ification): no right or wrong — anything CAN be a specification.

But to be useful, must: {be sufficiently strong (otherwise the shr-d obj can't be used in a prog); can be imple-d efficiently (otherwise remains a theory); often a trade-off}.

Example: mutex problem with $x = x + 1$ (2 processes). By saying the execution of "writing value (back) to register x to be 1 (finally)" is NOT GOOD, we implicitly assumed SOME consistency defn for "Integer ADT" must having value 2.

Why do we care about consistency? Mutex is actually a way to ensure consistency (based on some defn of consistency). Our goal in (this) lec is to {formalize such const-cy defn; explore some other useful const-cy defn-s}.

Sequential Consistency. First defined by Lamport: "the results ... is the same AS IF {operations of all the process(ors) were executed in some sequential order; AND the operations of each individual proc(essor) appear in THIS sequence [in the order specified by the program]}".

Abridged: {use sequential order as a comparison point; focus on RESULTS only (that is what users care about — what if we only allow sequential histories?); require program-order is preserved}. Need a lot of formalism to formalize the above.

Formalizing a Parallel System. Operation: a single invocation/response pair of a single method of asingle shared obj by a proc({e an operation; proc(e) is the invoking process; obj(e) is the obj; inv(e) is invocation event (start time); resp(e) is reply event (finish time)}. 2 invoc events are same if invoker, invokee, parameters are the same [$inv(p, read, X)$] (similarly for response: invoker, invokee, response [$resp(p, read, X, 1)$]).

(Execution) History: history H is a sequence of invoc-s and resp-s ordered by WALL CLOCK TIME. Details: {For any invoc in H, corresponding resp is required to be in H; each executino of a parallel system corresponds to a history, and vice versa}.

Sequential and Concurrent History. History H is sequential if any invoc is always IMMEDIATELY followed by its response (i.e. no interleaving). Otherwise, (H is) called concurrent.

A sequential history H is legal if {all resp-s satisfies sequential sematics of the data-type; seq-sematics: semantics you would get if there is only 1 proc accessing that data (type)}. Note: it is possible for a sequential history not to be legal (if response(s) violate the "logical expected response").

Subhistory. Process p's process subhistory of H ($H \mid p$): subsequence of all events of p, thus, a process subhistory is always sequential. Object o's obj subhistory of H ($H \mid o$): subsequence of all events of o.

Equivalency and Proc-Ord. 2 histories are equivalent if they have exactly same set of events: {same events imply all RESPONSES are the same; BUT ordering of events may be different (hence the *set* definition); user only cares about responses}. Proc-order is a partial order among all events: {for any 2 events of the same proc, proc-order is the same as execution order; no other additional orderings}.

Sequential Consistency. H is sequentially consistent if it's equivalent to some legal sequential history S that preserves proc-order. {Use (legal) sequential history as a comparison point; docus on results only (what if we only allow sequential histories?); require that prog-order be preserved}.

Motivation for Linearizability. Seq-const-cy arguably THE most widely used const-cy defn (e.g. almost all commercial databases, many multi-processors). BUT sometimes not strong enough.

More Formalism after here.

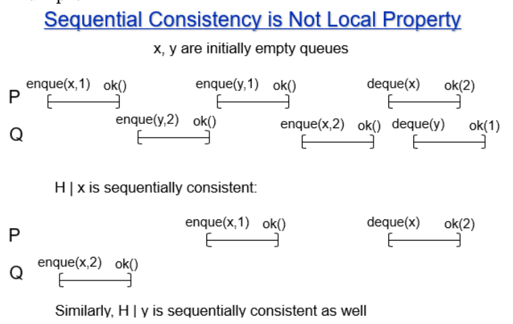
History H uniquely induces the following "<" partial order among operations. $o1 < o2$ iff response of $o1$ appears in H before invoc of $o2$. We call this *external order* (textbook calls this occured-before order). Question (to ponder): what is external order induced by a sequential history?

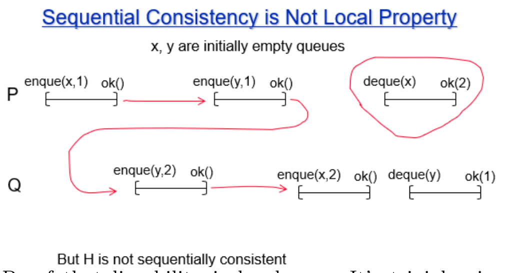
Linearizability. {Defn 1: execution is eequivalent to some execution so that [each operation happens instantaneous-ly at some point (called *linearization* point) between the invoc and resp]; Defn 2: History H is linearizable if {equivalent to some legal sequential history S, AND; S preserves external order in H — namely, the partial-order induced by H is a subset of the partial-order induced by S}}.

A linearizable history must be sequentially consistent (linearizability is arguably the strongest form of const-cy people have ever defined).

Local property: linearizability is a local property, in the sense that H is lin-able iff for any obj x, $H \mid x$ is lin-able. Sequential property is not a local property.

Example:


Sequential Consistency is Not Local Property
x, y are initially empty queues
P
enque(x,1) ok()
enque(y,1) ok()
deque(x) ok(2)
Q
enque(y,2) ok()
enque(x,2) ok()
deque(y) ok(1)
H | x is sequentially consistent:
P
enque(x,1) ok()
deque(x) ok(2)
Q
enque(x,2) ok()
Similarly, H | y is sequentially consistent as well


Sequential Consistency is Not Local Property
x, y are initially empty queues
P
enque(x,1) ok()
enque(y,1) ok()
deque(x) ok(2)
Q
enque(y,2) ok()
enque(x,2) ok()
deque(y) ok(1)
But H is not sequentially consistent

Proof that lin-ability is local prop. It's trivial using defn 1, proving using defn 2:

Proof. Construct directed graph among all operations as following (dir-edge created from $o1$ to $o2$ if) {$o1$, $o2$ is on same obj x, $o1$ is before $o2$ when linearizing $H \mid x$; OR $o1 < o2$ in external order (i.e. resp of $o1$ appears (is) before invoc of $o2$ IN H}. We say that {the first one is $o1 \to o2$ due to obj, the second one is due to H}.

Lemma is the resulting dir-graph is acyclic. Proof by contradiction. The cycle must be even-length-ed, and consists of (after merging same-cause-d edges by transitivity) edges due to *obj* and to H. Then we bash, directly arguing based on wall-clock-time intervals.

Const-cy defn-s for registers. Register is a kind of ADT: a single value that can be read and written.

(from stackoverflow: address is in comment after this)

Basic Idea: Reads and writes to a shared memory location take a finite period of time, so they may either overlap, or be completely distinct.

[wall of text starts here]

A safe register is only safe as far as reads that do not overlap writes. If a read does not overlap any writes then it must read the value written by the most recent write. Otherwise it may return any value that the register may hold. So, in thread 2, the second read must return the value written by the second write, but the first read can return any valid value.

A regular register adds the additional guarantee that if a read overlaps with a write then it will either read the old value or the new one, but multiple reads that overlap the write do not have to agree on which, and the value may appear to "flicker" back and forth. This means that two reads from the same thread (such as in thread 3 above) that both overlap the write may appear "out of order": the earlier read returning the new value, and the later returning the old value.

An atomic register guarantees that the reads and writes appears to happen at a single point in time. Readers that act at a point before that point will all read the old value and readers that act after that point will all read the new value. In particular, if two reads from the same thread overlap a write then the later read cannot return the old value if the earlier read returns the new one. Atomic registers are linearizable.

[Wall of text ends]

Safe not ensure sequential const-cy, and vice versa. Atomic must be regular, regular must be safe.

4 When physical clocks not available

Assumptions. Proc can perform 3 kinds of atomic actions/events: {local computation; send a single message to a single process; receive a single message from a single proc}. No atomic broadcast.

Communication model: {point-to-point; error-free + infinite buffer; potentially out of order}.

Motivation. Many protocols need to impose an ordering among events, e.g. {Event A: mom deposit some money into your bank account; Event B: the bank's ATM card is used to buy some stuff; bank needs to properly order the 2 events.} Physical clocks: {seems to (completely) solve the problem; BUT what about the theory of relativity?; even without it, there are efficiency problems}.

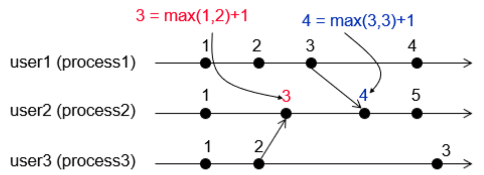
How accurate is needed/sufficient? Clock error needs to be less than message-prop-delay (in other words, it has to be "quiteäccurate).

Software "clocks": can incur much lower overhead than maintaining (sufficiently) accurate physical clocks. It allows a protocol to infer ordering (among events). Goal: capture event ordering (ofc!) that are visible to users (which users?) who DO NOT HAVE PHYSICAL CLOCKS.

But what orderings are visible to such users? Process-order, send-receive-order, transitivity.

"Happened-Before" relation. It captures the ordering that is visible to users WHEN there is NO PHYSICAL CLOCK. So, in short, goal of software "clockïs to capture the above "happened-before"relation.

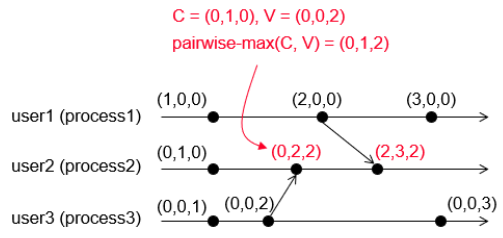
Logical clocks. {each proc has local counter C; increment C at each "local computationänd ßendëvent; when send message, logical clock value V attached to message; at each "receiveëvent, $C = \max(C, V) + 1$}.



Properties: s hap-bef $t \Rightarrow$ logic-clock$(s) <$ logic-clock(t).

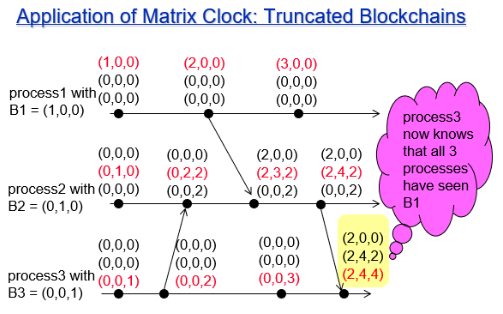
Vector clocks. $v_1 \le v_2$ (n integers as its vector clock value) if every field in $v1$ less than or equal to the corresponding field in $v2$. $v1 < v2$ iff $v1 \le v2$, $v1 \ne v2$.

Protocol. {each proc i has local vector C; increment $C[i]$ at each "local computationänd ßendëvent; when ßend"msg, vector clock value V attached to the msg; at "rcvëvent, C =pairwise-max(C, V) AND THEN $C[i]{+}{+}$}.



Properties: s hap-bef $t \Leftrightarrow$ vect-clock$(s) <$ vect-clock(t). Proof: if s send and t corresponding-receive on q, we have $VS[p] \le VT[p]$ since vect-clock$(s) <$ vect-clock(t) $\to$ must be a sequence of message from p to q after s, before t.

Matrix clocks. Motivation: {my vect-clock describe what I ßee"; in some app-s, I ALSO want to know what other people see (?!)}. Definitions: {each event has n vector clocks; ith vect-clock on proc i is called proc-i's *principle vector* (note: this thing is the same as vector clock before); non-principle vectors are "just"piggybacked on messages (too) to update "knowledge"}.



Protocol. 2 cases: principle vector C or non-principle vector C on process i (suppose it corresponds to process j). 1st case: same as before (with $C[i]{+}{+}$; but all n vectors are sent too). 2nd case: don't do $C[i]{+}{+}$, just do C =pairwise-max(C, V).

5 Snapshots "at same time"

Goal: picture of global computation, "taken at same time" (useful for debugging, backup/check-pointing [lecture: "if want (or forced to, e.g. node or system crash" to stop computation, can just continue from the snapshot's recorded state], calculating global predicate (i.e. how much currency do we have in country: notice money constantly flows among people)).

Motivation: if each proc have access to completely accurate physical time — then trivial. e.g. all proc-s record local state at 1:00:00 pm. 3 cases: either all record before message is on-the-fly, all record while msg is on-the-fly, or after message arrives (get \$200 total regardless of when snapshot is taken).

BUT don't have completely accurate phys-time (may record state in slightly different time). We can't tolerate much inaccuracy, since it NEEDS TO BE BELOW minimum-msg-prop-delay. Hard!

So we weaken our goal. Into, take snapshot of global comput, "a snapshot of local states on n proc-s such that the global snapshot COULD have happened SOMETIME in the PAST".

Details: {"in the past": we don't know when it occured (it may not have occured at the time specified by global snapshot); "could have happened", user cannot tell the difference (again, in reality, may not have happened; BUT, in an alternate universe where computation occured in that (exact) (happened-before) order, it COULD happen)}.

Note: perhaps surprisingly, such a weaker goal is still useful (for our three purposes earlier, that is described on the first paragraph).

Terminology: 3 kinds of snapshots. {Consistent snapshot: snapshot of local states on n proc-s s.t. global snapshot could have happened sometime in the past; global snapshot: a set of events s.t. if $e2$ is in the set AND $e1$ is before $e2$ IN PROCESS ORDER (not in hap-bef order), then $e1$ must be in set; C-G-S: global snapshot THAT satisfies "$e2$ in set AND $e1 < e2$ in snd-rcv order $\to$ $e1$ in set"}.

Note that in C-G-S, we don't need to include transitive relations, since the entire notion is captured in those two orders.

Existence of C-G-S. A good habit to prove the existence of something newly defined. (Why? So that our object is useful and not (only) a fantasy and theory; what use is it if we work with objects that does not exist?)

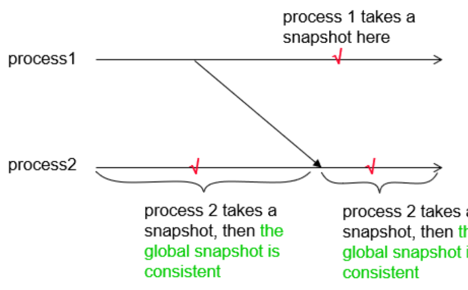
Theorem: consider the ordered events $e1, e2, \ldots$ on any given proc; for any positive int m, there exists a C-G-S S s.t. $\{e_i \in S$, for $i \le m$; $e_i \notin S$, for $i > m\}$. Proof sketch: color events must be in S blue, must not be in S red, and note that those two are mutually exclusive.

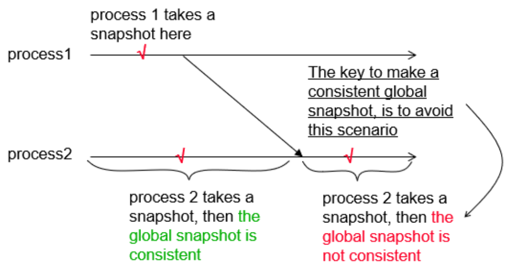
Hard-to-understand protocol. Especially the four case bash

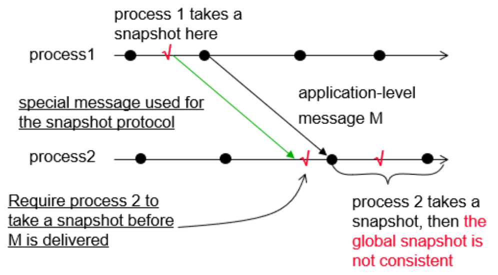
(TODO: need to re-review this again and write my conclusions! DONE: actually, after midts, and going back and forth between last slides and the four conditions, it's understandable)

Premises. {Communication model: {no msg loss; comm-channels are unidirectional; FIFO delivery on each channel}; ensuring FIFO: {each proc maintains a msg-num-counter for each channel AND stamps each message sent; receiver will only deliver messages (to itself) in order}}.

Protocol Intuition:

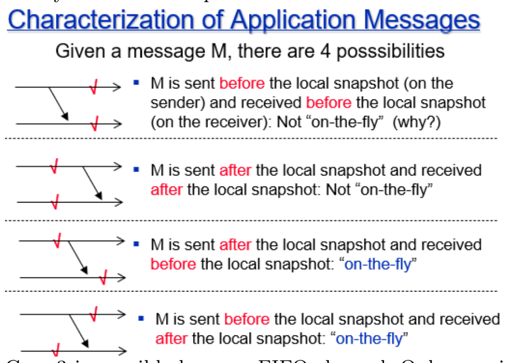

process 1 takes a snapshot here
process1
process2
process 2 takes a snapshot, then the global snapshot is consistent
process 2 takes a snapshot, then the global snapshot is consistent


process 1 takes a snapshot here
process1
The key to make a consistent global snapshot, is to avoid this scenario
process2
process 2 takes a snapshot, then the global snapshot is consistent
process 2 takes a snapshot, then the global snapshot is not consistent


process 1 takes a snapshot here
process1
application-level message M
special message used for the snapshot protocol
process2
Require process 2 to take a snapshot before M is delivered
process 2 takes a snapshot, then the global snapshot is not consistent

Chandy-Lamport's Protocol: taking loc-snapsh-s. {1. Each process is either: {Red (it has taken the local snapshot), OR; White (it has not taken the local snapshot)}; 2. protocol (is (to be)) initiated by a single process, by turning itself from White too Red: {Once a process turns to Red, it immediately send out Marker msgs to all other proc-s; upon receiving Marker, a proc turns Red; there will be a total of $n*(n-1)$ Marker msg-s}}.

Next, we would like to also capture msg-s that are "on-the-fly" when the snapshot is taken.

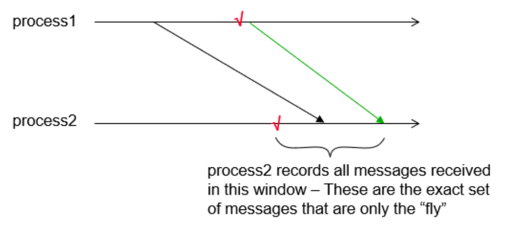


Characterization of Application Messages

Given a message M, there are 4 posssibilities

- M is sent before the local snapshot (on the sender) and received before the local snapshot (on the receiver): Not "on-the-fly" (why?)
- M is sent after the local snapshot and received after the local snapshot: Not "on-the-fly"
- M is sent after the local snapshot and received before the local snapshot: "on-the-fly"
- M is sent before the local snapshot and received after the local snapshot: "on-the-fly"

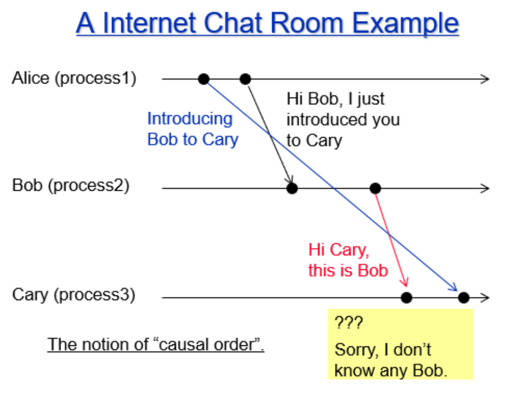
Case 3 impossible because FIFO channel. Only remaining case: (is) case 4. For that case, the protocol takes snapshots for messages on the fly (as follows):


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

6 The thing I f-ed up in midterms

Side note: also probably the most technical concepts and protocol with longest correctness proof.

Example:


A Internet Chat Room Example
Alice (process1)
Introducing Bob to Cary
Hi Bob, I just introduced you to Cary
Bob (process2)
Hi Cary, this is Bob
Cary (process3)
The notion of "causal order".
???
Sorry, I don't know any Bob.

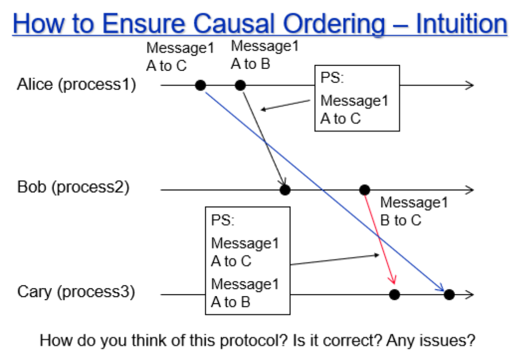
Formalizing Notion of Causal Order. If a "send" event $s1$ *caused* a "send" event $s2$: {*Causal* order requires the corresponding "receive" event $r1$ be before $r2$ (IF $r1$, $r2$ on same process); BUT how do we know whether $s1$ CAUSED $s2$?}

Note: $a \to b$ means hap-bef and $a < b$ necessarily on same process.

(Here's when I was wrong and didn't understand the motivation)

We generalize it (and "play it safe", if a "send" event $s1$ *happened before* a "send" event $s2$: {then $s1$ MAY have caused $s2$; let's be pessimistic and safe — assume $s1$ indeed caused $s2$ (can we avoid being pessimistic?)}

Causal order: If $s1$ hap-bef (happened-before) $s2$, and "$r1$ and $r2$ are on the same process", THEN $r1$ MUST be before $r2$.


How to Ensure Causal Ordering – Intuition
Message1 A to C
Message1 A to B
Alice (process1)
PS: Message1 A to C
Bob (process2)
Message1 B to C
PS: Message1 A to C Message1 A to B
Cary (process3)
How do you think of this protocol? Is it correct? Any issues?

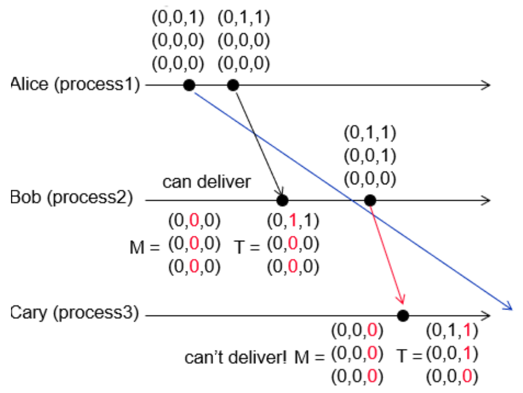
Answer to "is this a correct protocol" from lecture: "YES, it is correct, BUT it is a waste; no need to send entire log, just send timestamp (i.e. order) and number of messages sent". Response to that: "I think this is a wonderful answer" - Prof Yu Haifeng.

Protocol on How to Ensure Causal Ordering: {Details. each proc maintains n by n matrix M: {not the matrix clock!; On send. $M[i,j]$: # of msg-s sent from i to j, as known by proc i}; if proc i send a msg to proc j, {on proc i: $M[i,j]++$; piggyback M on the msg}}.

Continued. {On receive message. Upon proc j rcv-ing msg from proc i (with matrix T piggybacked), let M be local matrix on proc j. Deliver the msg (to itself) AND set M =pairwise-max(M,T), if $\{T[k.j] \le M[k,j]$ for all $k \neq i$; $T[i,j] = M[k,j]+1\}$. INTUITIVELY, $M[i,j]$ ON process j takes consecutive values; What happens otherwise? Delay message.}

Intuition for whole protocol: matrix summarizes tjhe info in msg-log in the previous protocol on Slide 6.

Example run of protocol:



Summary of correctness proof: {1. All msg-s will eventually be delivered; 2. the delivery order satisfies causal order.}

Broadcast Messages. (this should be self-explanatory) but broadcast := every msg sent to all people (including sender itself), and modeled as n point-to-point msg-s.

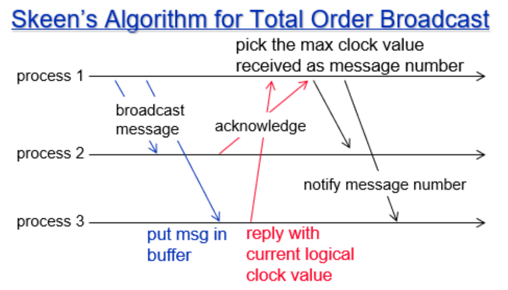
Causal Ordering of Br-Msg-s: each process uses the previous protocol (for same reason as before, example application: internet chat room).

Total Ordering of Br-Msg-s: all messages delivered to all proc-s in exactly the same order (also called atomic broadcast). This total ordering ONLY applies to broadcast msg-s.

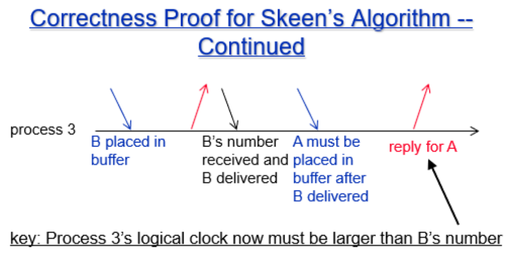
Note that those two not imply each other. Example of causal but not total: 2 proc-s, either proc-s send msg to itself with 1 time-unit prop-delay, and to the other proc with 100 time-unit prop-delay. Clearly, messages not received with the same ordering (if received "raw", without any buffer).

Coordinator for Total-Order Broadcast. Here, a special proc is assigned as the coordinator. Protocol (to broadcast a msg): {1. send msg to coordinator; 2. coordinator assigns a seq num to the msg; 3. coordinator forward the msg to all proc-s with the seq-num; 4. msg-s delivered according to seq-num order}. Problem with this: coordinator has too much control.

Skeen's Algo. {Each proc maintains: a log-clock and msg-buffer for undelivered msg-s; A msg (in the buffer) is delivered (or removed) IF {all msg-s in the buffer have been assigned num-s; and this msg (the one that is being removed) has the smallest number}}. The protocol itself is described in the following:


Skeen's Algorithm for Total Order Broadcast
pick the max clock value received as message number
process 1
broadcast message
acknowledge
process 2
notify message number
process 3
put msg in buffer
reply with current logical clock value

Correctness Proof. 3 Claims: {all msg-s will be assigned msg-num-s; all msg-s will be delivered (eventually!); if msg A has a number smaller than B, then B is delivered after A (proof is by contradiction, trivial (??? by trivial, slides mean it's really not that hard and not supposed to be hard))}.


Correctness Proof for Skeen's Algorithm -- Continued
process 3
B placed in buffer
B's number received and B delivered
A must be placed in buffer after B delivered
reply for A
key: Process 3's logical clock now must be larger than B's number

7 This topic has 3230 (and 2040S) vibes.

The topic is leader election on a ring AND leader election on a general graph.

Motivation: {we often need a coordinator in distributed systems (leader, distinguished node/process); if we have a leader, mutex is trivially solved: {leader determines who enters CS; but this is an inefficient solution}; if we have a leader, total-ordered-broadcast trivially solved (i.e. leader stamps msg-s with consecutive integers)}.

Leader Election on Anonymous Ring. Impossible using deterministic algo-s. Reason: {same initial state; algo on each node same; each step is same; final state is same (i.e. logically, all will be leaders by these kind of algos)}.

L-E on Ring: Chang-Roberts Algo. Constraints: {1. each node has unique id; 2. nodes only send msg-s clockwise; 3. each node acts on its own}. Protocol: {a node send election msg with its own id clockwise; election msg is forwarded iff id in msg is larger than own msg (otherwise, msg is discarded)}. A node becomes leader if it sees its own election msg.

Proof that avg performance is $O(n \log n)$. Average: taken over all possible orderings of the nodes on the ring, with each ordering having same probability. (this is CS2040S stuffs).

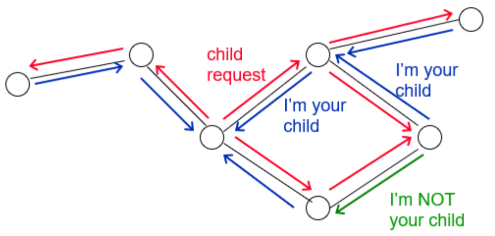
Define random variable x_k := num of msg-s caused by election msg from node k (node k := node with k^{th} smallest id).

Then, we have $P[x_k = i (\leq k) \mid x_k > i-1$ is $\frac{n-k}{n-i}$. Further simplify this by defining $p := \frac{n-k}{n-1}$, so all $P[x_k = i \mid x_k > i-1] \geq p$. Then, it's just $< n + \sum_{1 \leq k \leq n-1} \frac{n-1}{n-k}$ which is $O(n \log n)$.

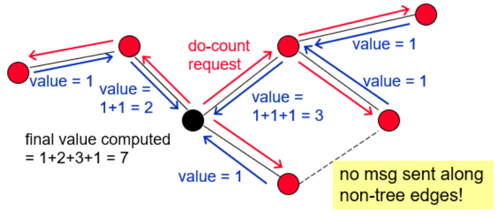
L-E on General Graph. If n known, on complete graph: {each node sends id to all other nodes; wait until you receive n ids; biggest id wins}, on (any) connected graph: {*flood* your id to all other nodes (how? protocol ensuring each node passes un-received or un-acknowledged ids to neighbors); wait until you receive n ids; biggest id wins}.

Now, "problem" is when n unknown. In complete graph, n must be known! On any connected graph: we "construct/describe" a(n auxiliary) protocol to calculate the number of nodes. This protocol is initiated by ANY node who wants to know n (essentially, it establishes a span-tree starting (i.e. rooted) from the initiator).

Spanning-Tree Construction. Important points: {remember: no centralized coordinator; goal of the protocol: each node knows its parent and children (i.e. a distributed tree)}. Protocol illustration:



This is after constructing the span-tree, then everyone count nodes (in their subtree) and passes to parent. Illustration:



Note: this span-tree is also useful for other purposes: {broadcast: root not need to send out n msg-s; aggregation: e.g. avg-temperature sensed by distributed sensors; min or max calculation}.

8 Distributed Consensus: the easier cases

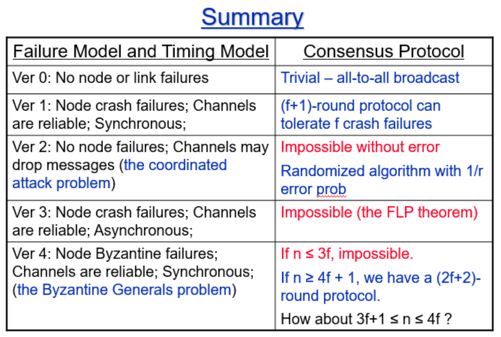
A note on homework's impossibility result.

The nodes are identical, so any randomized algo performed on such identical nodes will send THE SAME MESSAGE every round (of passing messages). So, when P1 in ring 1 sends message m to the next node (which is itself), this execution is INDISTINGUISHABLE from ring 2's execution, where ring 2's P2 will (also) send m back to P1.

Indist-able notion: is to make rigorous by what we mean by "cannot differentiate".

For known ring sizes: an easier formalization is to eliminate losers, and prove that it has at least one loser "eventually" (with probability approaching 1).

Summary of all 5 (case 0 is trivial, remaining 4 cases covered in two lectures):



Summary

Failure Model and Timing Model	Consensus Protocol
Ver 0: No node or link failures	Trivial – all-to-all broadcast
Ver 1: Node crash failures; Channels are reliable; Synchronous;	(f+1)-round protocol can tolerate f crash failures
Ver 2: No node failures; Channels may drop messages (the coordinated attack problem)	Impossible without error Randomized algorithm with 1/r error prob
Ver 3: Node crash failures; Channels are reliable; Asynchronous;	Impossible (the FLP theorem)
Ver 4: Node Byzantine failures; Channels are reliable; Synchronous; (the Byzantine Generals problem)	If n ≤ 3f, impossible. If n ≥ 4f + 1, we have a (2f+2)-round protocol. How about 3f+1 ≤ n ≤ 4f ?

Motivation (enlightening, important). When building distributed systems, a lot of the times, the problem is about reaching consensus (e.g. whether a flight ticket should be purchased). And we want to create protocols that tolerate failures (communication failures, system/-node failures; harder problem: failures are "ongoing").

"I'm sure things will be different when they fail. Question is: <how will they fail?>" Some messages are sent but not all (i.e. network/node failures will partition the process into multiple disconnected components)

So, need to be rigorous, as solutions (and the possibility of solutions) vary wildly across different cases (cases: failure model and timing model).

Node failure: process starts execution and in the middle of execution, the process stops execution ("abruptly") (i.e. it does not send additional messages to other processes; different from Windows OS crash where it may still send messages to other processes before finally stops execution (for all processes)).

Note on timing model: synchronous timing model means message delivery and node processing times have known upper bounds. This means, these parameters are passed into your dist-sys-algos and the protocol can use them (to do things: e.g. to see whether node response is still within a reasonable time, the term is "accurate failure detection")

Goal. {Termination: all nodes (that have not failed; for those that have failed, nothing we can do about them) eventually decide; agreement: all nodes that decide (??? including those who later crash [and "cannot be contacted anymore"]) should decide on same value;}

Abstraction of rounds. "inter-locked round: {does some local-comput; sends 1 msg to every other proc (sends 1 msg $n-1$ times, msg-s may be different across different proc-s); rcv-s 1 msg from every other proc (so total receives $n-1$ msg-s)}".

This abstraction is useful because "all proc-s make progress every round". Also useful because every sending-out-msg-s can be (conceptually) viewed as a sequence ever round. Also can be used for accurate failure detection (lack of message will certainly indicate failure).

Protocol (1) Intuition. Proceeding from the "no failure" case, we pick max of (hopefully) n messages. If fail, however, a process MAY send inputs to ONLY a SUBSET of the proc-s. Solution: "keep forwarding the values" (i.e. send all input values that it HAS seen).

The protocol needs <max number of failures f (the way I see it, once a process crashes, it "no longer has the ability to rejoin the "conversation"")> as an input. Well, so we have f+1 rounds to tolerate these f failures.

Protocol (code):

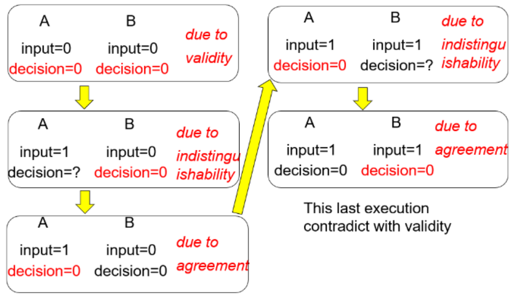
```
S <- {my input}
for (int i = 1; i <= f+1; i++) {
   // do this for f+1 rounds
   send S to all other nodes
   receive n-1 sets;
   for each set T received: S <- Union(S, T)
}
Decide on min(S);
```

Proof. One good round in (any of) the $f + 1$ rounds must exist. When that round is over, ALL non-crashing nodes will have "same S; i.e. same sets of (node) inputs". (additional note: after that rounds, S values won't change)

Note: can't do better than $f + 1$ rounds (proof beyond this course). Deterministic case is homework: easy, just set $f = 1$.

Version 2. The 3 goals still the same. Impossible to reach above goal by using deterministic algo, since channel may drop all messages.

Rigorous Proof (technique is important since this technique will be used in more sophisticated/complicated/complex cases). Prior defn: execution α, β INDISTINGUISHABLE for some node if node sees "same things" (i.e. msg-s and input-s) in both executions.



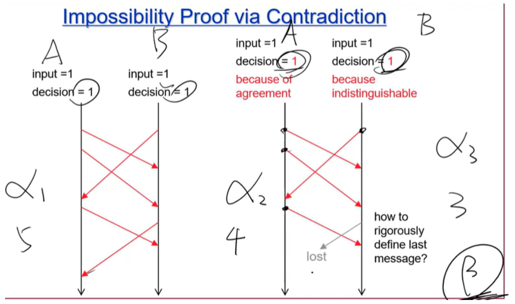
Proof details: in β (i.e. second execution), let B not receive any message from A (comm-channel drops all msg-s). In more detail: comm-channels ALSO drops all msg-s in α (1st execution); but the decisions are fixed in α due to validity (constraint). For the third "picture", it is still describing execution β. Recalling that this is a dist-consensus-algo, A must also decide on 0. (due to agreement). Fourth picture describes a third execution on γ due to indistinguish-able-ity (and same algorithm (!)). Fifth picture is analogue of third picture (where agreement "forces" B to conclude 0).

An answer to a question: we don't look at validity alone; we look at "whole package" that consists of {validity + termination + agreement + (additional constraint, in this case of Ver 2,) "deterministic".

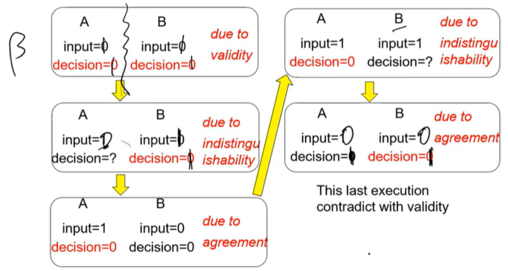
We see that previous problem is impossible to solve.

We weaken the constraints: {keep first 2 goals; weakened validity: {if all nodes start with 0, decision should be 0; if all start with 1 AND NO MSG IS LOST THROUGHOUT EXECUTION [i.e. channel is nice], decision should be 1; otherwise nodes are allowed to decide on anything [BUT, still must decide]}}.

Key point of proof (only explained in Lecture): is that there is no further acknowledgement after last message sent ("last" as seen by global time) + we can generalize until we get execution where all messages are dropped.



Then, we repeat same argument (as earlier).

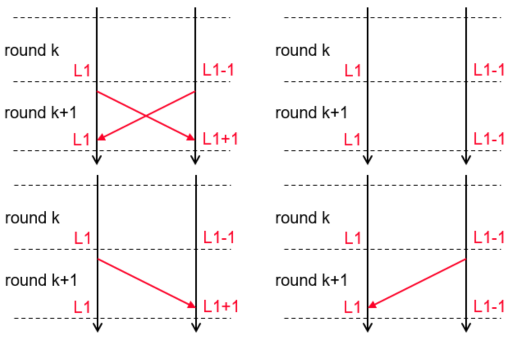


Allowing limited disagreement (example: airline refund funds, but want to keep it to a minimum). Goal: {keep first goal; all nodes decide on same value with probability of (1-*error*_*prob*); keep last goal}.

Example where assumption of [independence of coin-flips and message loss-es] is violated: it can be that when a process flips a coins (of zeroes and ones) and send it to another process, the sending of zeroes (say, in the physical layer, depending on how 0s and 1s are transmitted) is much simpler (hence, less error-prone) to sending ones. In this case, the message loss is correlated to the sent message.

Protocol-to-maintain-level algo: {P1 sends msg to P2 each round; P2 sends msg to P1 each round; Bar, input and current level attached on all msg-s; upon P2 receiving a msg with L1 attached: P2 sets L2 = L1 + 1; L1 is maintained similarly}.

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



// Note to self: FOR NEXT TIME, don't forget to use backtick backtick apostrophe apostrophe for quotes in LaTeX and use actual-double-quotes in markdown.

9 W9 Lecture.

Prof's remarks and explanation regarding previous lecture's ending.

Whether or not channel drops messages, protocol SHOULD DECIDE on SOMETHING. About "error of not achieving agreement", we should reason about [the ability of the randomized-algo] in the "worst-case of message loss".

This protocol is designed in a way such that no matter what the adversary does, $|L_1 - L_2| \leq 1$.

When can an error occur? If "bar happens to split L_1 and L_2". Concrete illustration: "if my midterm and my friend's midterm differs by 1, most likely, both fail or both pass; the pass threshold is unlikely to be between them".

How do we make this intuition rigorous? If there is a disagreement, one process decided on 1 and the other, on 0. Note that in this case, we must have $L_1 \neq L_2$. We divide into 3 cases: these are how we should think about it.

{Case 1: P1 sees P2 input, BUT P2 not see P1 input (hence, not see bar). This will imply $L_1 = 1, L_2 = 0$ (and bar needs to be set to 1); Case 2: same thing, but reversed; we need bar to also == 1; Case 3: P1 and P2 see each other's inputs (and hence, P2 sees bar). Error is when bar = $\max(L_1, L_2)$}.

Since the 3 cases are mutually exclusive, we have *error* $\leq \frac{1}{r}$. In other words, with $\geq 1 - \frac{1}{r}$ probability, agreement will happen.

Note: proof of "validity goal": no message loss $\Rightarrow L_1 = L_2$ $\Rightarrow$ must decide on 1.

Now we get to FLP. First, we summarize the lecture notes.

Version 3. Failure model: {nodes may experience (crash) failures; comm-channels are reliable}. Timing model: asynchronous. Means process delay and message delay are finite but unbounded — this means you can't find a bound s.t. all msg delays are below that bound (in practice, can be msg-s delayed for a long time), so, can no longer define a round (if we don't receive a msg for a long time, we don't know if [the sender has failed] or [the message is just delayed]).

Goal: every one of the 3 goals. FLP thm (Fischer, Lynch, Paterson, '85), impossible to solve even with a single node crash failure. Fundamental reason: protocol is unable to accurately detect node failure.

FORMALISMS for proof. GOAL: ABSTRACT THE EXECUTION of any possible deterministic protocol.

Process have some local state (whatever they are: unabstracted, they are like CS2106 OS/machine context etc) and two special variables *input* $\in \{0, 1\}$ and *decision* $\in \{0, 1\}$. *decicion* initially null and can be written exactly once.

Comm-channel: set of msg-s "on-the-fly". This msg-system captures state of all comm-channels: $\{(p, m) \mid$ msg m on-the-fly to proc $p\}$; all msg-s are distinct.

Send := add (*dest*, *content*) to msg-system. Receive (when invoked by proc p) := do 1 of 2 things, {1. remove some (p, *content*) from msg-system + return content, OR; 2. leave msg-system untouched (i.e. do nothing) + return null}.

Additional points (covered in Lecture, see below in "Lecture" section): out-of-order (i.e. no ordering imposed) or FIFO? Non-blocking receive or blocking receive?

Execution. (any) protocol execution can be abstracted to be an infinite sequence of events. (this abstraction of "infinite" is necessary to properly define (and handle/take-into-account) faulty proc-s).

Schedule σ is a seq of events that captures the execution of some protocol. Details: {σ can be applied to G if events can be applied to G in σ's order; $G' = \sigma(G)$ means if we apply σ to G, will end up with G'; not all $\sigma(G)$ is legal, since σ may actually not be applied to G}.

Reachable states from a particular state. G_2 reachable from G_1 if exists sched σ (of a protocol A, but the protocol is essentially not given a name since we always fix a protocol) such that $G_2 = \sigma(G_1)$.

Now for the Lecture explanation, transcribed into written notes.

Note on asynchronous-ity. Is it the case that "async timing model" then uninteresting (msg-s will eventually be delivered in current internet, right?) $\Rightarrow$ subtle notion that needs to be scrutinized closely.

Even if we impose upper bound of 1 month, it's impractical to set round equal to 1 month (so the synchronous case is just not applicable, and we need to "worry" about not-synchronous case too). Furthermore, protocol of $f+1$ rounds will last $f+1$ months, and this is just NUH-UH NO (extreme case, better not be modeled as synchro-timing-model).

Additional note on synchro/asynchro. No point to ask if "current system is sync/async" (since it's not a dichotomy; a system may not belong to either categories). Example 1: upper bound is unknown, example 2: upper bound is transient and volatile-ly changes as time goes (or if it's first async, then sync).

Key implication (as written in L-Notes): no longer can define a round (anymore), can't conclude that a crash must've happened if delay is very long (or "longer than normal").

Note on impossibility. Impossibility proofs are strong proofs: for all protocols it must hold. "i.e. if we consider a msg m sent by proc p to prove an impossibility result: what if some other protocol does not let p send that msg?"

Note on rigor (especially since previous note is on impossibility). Rigor helps clarify our intent so we don't end up with wrong answers (people wrote yes and gave proof for Q10 of midt, and wrote no and gave code for Q11 of midt". Rigor helps counter-intuitive results to be formalized and intuit (and this fact is often counter-intuitive since it seemed to diminish the value of intuition).

Note on why the abstractions are the way they are. We need abstraction that have "(all) necessary details, that is sufficiently complex to capture the whole situation".

Note on msg-system. A state is a representative of "when the snapshot of on-the-fly-msg-s are taken, in a particular time". Note that we can add seq-num-s to distinguish ALL msg-s (without impacting execution).

Note on receive protocol. When it does nothing (telling a particular process p that no msg has arrived), it represents one of two cases: {1. no msg for p, or; 2. there is a msg for p, but still flying}.

Note on the FIFO/out-of-order (and blocking/nonblocking rcv). Can't say "FIFO special case of out-of-order" (since no-node-failure $\subset$ $\leq$ 1 node failure, and the latter being impossible does not imply the first is impossible, since the first cannot be used to implement the latter — this is 3230 reduction technique).

So, we need to say that IF an algo A exist that ensure correctness using FIFO ordering, I construct algo B that invoke A and DO EXACTLY SAME THING, without FIFO ordering (well, we make B's behavior mimic A by attaching sequence numbers). Similar reasoning for blocking/non-blocking.

Fundamental order of "FIFO being impossible" is that out-of-order protocols can implement FIFO. i.e. these are not "just minor technical details".

Note on events. We define it that way (transitioning from one glob-state to another glob-state), rather than in terms of machine-level instructions, as {1. 1 or 2 (or any number) of M-L-instr-s may not change "too much"; 2. It's just about sending and receiving msg-s that are not disturbed/influenced by other proc-s}.

Note on why "infinite sequence of events" are necessary. Note that any sequence of partial-order-msg-s can be converted into a infinite-sequence-total-order of msg-s (i.e. no longer viewed as interleaving n-process execution but as a seq-of-events with total-order). To keep it infinite (especially after decision), can just invoke p receive null (to artificially elongate the sequence).

Why look at infinite? We don't know which process has failed (maybe process 1 not fail yet, when it becomes "very slow"?) How does this notion help us? If process crash, this process must only take finite steps; so it helps "capture notion of [failing or slowed] (i.e. asynchronous) accurately".

Note on scheduling. If protocol has incurred e_1, e_2, e_3 on global state G, we say $e_3 e_2 e_1 (G_0) = G_3$ where $e_i(G_{i-1}) = G_i$ (and we need e_i to be applic-able to G_{i-1}).

10 LMAO Have fun (actually not THAT difficult)

It's just long. Reread the L-Notes multiple times, back and forth, to get it.

I don't have the willingness to rewrite and regurgitate everything here; I already did it in my handwritten notes. I'll just rewrite some subtle notions here.

PROOF (Lecture Notes). General proof technique: {(we) act as adversary to defeat the consensus-protocol; (we) scheduler can pick which msg-s to deliver + which msgs-s will "take next step" (under constraint of async: delay still finite); "actual" goal is to prevent protocol from "ever deciding"}.

Classification of global states: state G is {0-valent iff 0 is the only possible decision reachable from G (may/-may not YET decided on 0, but if not, eventually WILL and MUST decide on 0); 1-valent; univalent := 0-valent $\cup$ 1-valent; bivalent := !univalent}.

Note on Lemma 2. Process executing σ_1 disjoint from process executing σ_2: this is saying that $P_3 P_4$'s msg-s going first and $P_1 P_2$'s msg-s going first makes no difference. "Interchange ordering won't change outcome", since "the messages do not interact with each other". Also, we do not allow msg-s to execute between σ_1, σ_2 (or vice versa); so the sequences can be applied to form $\sigma_2(\sigma_1(G))$, or vice versa.

"The key assumptions of this Lemma is hidden in its formulation".

Note on Base Case of Lemma 2. $(G_1 =)$ $e_1(e_2(G))$ = $e_2(e_1(G))$ $(= G_2)$. (I made a note that sender do not need to be specified as an additional parameter; it can be attached to the msg m itself, and we also need to differentiate the process receiving it. In other words, m can contain "everything else" besides the process it is intended for).

Note on Lemma 4. $e(\sigma(G))$ with $|\sigma|$ finite means the delay of msg e is finite (but, the constraint of unboundedness (execution delay) is natural too since we cannot put a bound on σ). In other words, scheduler "plays by the rules" (i.e. it does not drop msg-s, and all msg-s are delivered, and all nonfaulty proc-s take infinite num of steps (instead of faulty ones that terminate/crash after finite num of steps)).

Note on Lem 4 - Claim 2. The prefix of schedule σ is from the back, since the back is applied first to G.

Lem 4 - Claim 3. "even in the most problematic case, can set $F_0 = G$ and $F_1 = d(G)$ where d is first event that takes G to G_1, with G_1 being 1-valent".

// NEED TO ACTUALLY REVIEW THIS Note on remaining proof for Lem 4. Protocol hangs on one process p, when protocol is supposed to tolerate one crash failure. i.e. the global state F_0 is very strange: "shaky, entirely dependent/determined by p's ordering (which comes first) $\rightarrow$ AND most other "critical" (decision-altering) events is ALSO on p".

We name the states F_0 and F_1 ("name" carried over (inherited) from Claim 4) s.t. $e(F_0)$ is 0-valent, $d(F_0)$ is F_1 and $e(F_1)$ is 1-valent. (this is WLOG, otherwise, fix a F_i and F_{1-i} i-valent and $(1-i)$-valent).

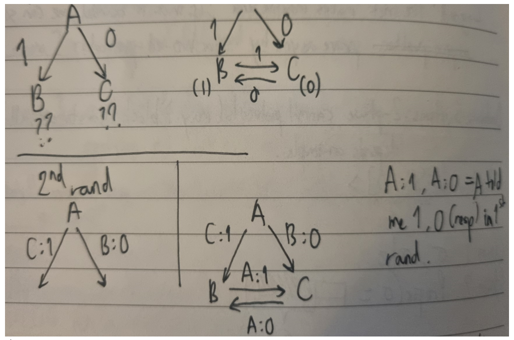
We must have e, d on same process; otherwise 0-valent state (i.e. $e(F_0)$) $\Rightarrow$ 1-valent state (i.e. $e(F_1)$).

Then consider the set {all exec-s starting from F_0}. There are 3-layered constraints (i.e. there is an execution in which {1. some process decides; 2. process p not execute any steps; 2'. since protocol must be able to tolerate 1 process-failure (such a protocol's execution must be that which p not execute steps/not take any steps)}.

Let we have $F_0 \rightarrow_\sigma T$; σ is applied to F_0 to give T, a state in which one process has decided. However, e has not been applied; so any decision that has been taken is "void" (i.e. decision cannot be 0, otherwise contradict with the fact that $e(d(T))$ is 1-valent, and cannot be 1, otherwise contradict with $e(T)$ being 0-valent).

11 Byzantine Failures AND Intro to Self-stabilization

Byzantine Protocol.



(note, in the handwritten media, "resp" means "respectively", not "response").

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
round for          send V[i] to all processes;
all-to-all         set V[1..n] to be the n values received;
broadcast          if (value x occurs (> n/2) times in V) proposal = x;
                   else proposal = 0;

coordinator        if (k==i) send proposal to all; // I am coordinator
round              receive coordinatorProposal from the coordinator

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

Abridged version of code (+ intuition). Intuition: {what if last phase has faulty coordinator?; avoiding faulty coordinator by non-faulty process keeping their values; when to avoid? we do not listen if [see a lot of identical values from other proc-s, with each phase also having an all-to-all broadcast round]}.

Code. two "phases" (ofc, after broadcast round): {1st phase only done by coordinator: it sends proposal to (all) proc-s; if coordinator is non-faulty, all proc-s sees proposal (consensus established by coordinator!); deciding phase is a phase with nonfaulty coordinator} and {2nd phase: is essentially the above intuition}.

When does Byzantine failure occur/what use case does it have? "worse than crashing" behavior: {confusing for other honest protocols (security settings: crypto-s deal with byzantine failures); very large scale of data integrity and precision (proc-s may behave in unpredictable ways due to strange hardware issues, i.e. "they can do anything they want", we do not want to care about the initial values of those nodes; so we really need to consider the very worst behavior of such bad nodes}.

Note that the example earlier is not a proof for unsolvable-ness (even for $f == 1$), since the scenario only shows the impossibleness of ONE protocol (what about other protocols? Especially a protocol with 10 rounds; or even other protocols with 2 rounds (!)).

Why the word Byzantine? Generals deciding to attack or to not attack (a "toy story") ⇒ generals can talk via phone lines (but not zoom meeting with everyone) + some are traders and have been bribed by the enemy to defeat their own boss. And the good generals try to achieve agreement.

Post-hoc + rough summary of protocol (usage and correctness). $V[i]$ respresents constantly updating "running-max/current-decision" → what if the inputs "are camouflaged/overriden"; if all values are 1 at beginning, is there a possibility of "initially 1-agreement or 1-valent state" being overshadowed by any-other-agreement after initial state? The cases seem overwhelming.

When does byzantine faults occur; can they occur while protocol is running (i.e. adaptive), or does adversary pick $f+1$ before protocol starts (i.e. passive/"possible-to-be-WLOG-ed").

Intuition of usage(s) of the two Lemmas: Lemma 1 holds to prevent round-1 (and, of course, subsequent rounds) jumbling-up-inputs (that are established at round-0; i.e. before protocol starts) by faulty coordinator.

Lemma 2 helps ensures (artificial) agreement to be made if not all non-faulty proc-s hold the same initial input.

Proof-of-correctness given Lemma 1's ∩ Lemma 2's correctness: {Termination: obvious; Validity (that is, if all have same inputs initially, the decision must be that value; otherwise, can decide on anything EXCEPT must satisfy agreement constraint (of course!)): Lemma 1 (without invoking Lemma 2); Agreement: proof involved slightly more contrived logic, {1. exists ≥ 1 deciding phase(s); 2. (invoking Lemma 2) after deciding phase, all non-faulty proc-s $V[i]$ have same value; 3. (invoking Lemma 1) $V[i]$ on nonfaulty not change}}.

Lemma 1's proof is obvious. Threshold is $\frac{n}{2}+f$, we have $\frac{n}{2}+f < n-f$ (any $n-f$ non-faulty will withstand the threshold).

Lemma 2's proof (slightly more complicated): 2 cases, as shown in the two media-s below.

- Case 1: Coordinator has proposal = x; (x must be unique on coordinator)
  - On coordinator: x appears (>n/2) times in V ⇒ (>n/2-f) must be from nonfaulty processes
  - On any other process: x appears (>n/2-f) times in V ⇒ Impossible for y to appear (>n/2+f) times in V

```
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) proposal = x;
        else proposal = 0;

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

        if (value y occurs (> n/2 + f) times in V) V[i] = y;
        else V[i] = coordinatorProposal; }
```

1. if proposal = x, (+ the fact that coordinator is non-faulty), then x appears $> n/2$ times (and this x is unique). Any other value $y \neq x$ cannot appear $> n/2+f$ times (on any other process).

- Case 2: Coordinator has proposal = 0;
  - On coordinator: no value x appears (>n/2) times in V
  - On any other process: Impossible for y to appear (>n/2+f) times in V

```
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) proposal = x;
        else proposal = 0;

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

        if (value y occurs (> n/2 + f) times in V) V[i] = y;
        else V[i] = coordinatorProposal;
}
```

2. if proposal = 0, no value appear $> n/2$ times on coordinator; SO, no value can appear $> n/2+f$ on any-other-proc's V.

Stabilization.

Motivation. This notion is useful in data-streaming (i.e. video streaming) since it is single-source, and distributed to multiple destinations.

This is multicast (i.e. network has multiple layers) where the source does not port to everyone; since source (bitTorrent, not the big ones like Netflix) won't have enough bandwidth to broadcast to the entire system tree.

The tree need to be rebuilt if some node fails (the state changes from "legal" to "illegal"). Note that legal/illegal depends on application semantics.

Definition. A distributed algo is called $self-stabilising$ if no-more-faults (after a specified time period) implies the algorithm can repair itself (i.e. transition back from an illegal state to a legal state); and, once the system is legal, can ONLY transition to other legal states (i.e. the algo does not put itself into trouble, if trouble has not come). Note: this property only guaranteed when "the network is nice and not disrupt algo execution".

Of course, a next question that we can ask is "how much time is required to repair: 1 minute or 1 hour?".

Note: the notion of "wrong" is the value of the tuple (state) from/with-respect-to all other nodes (not just their neighbor(s)).

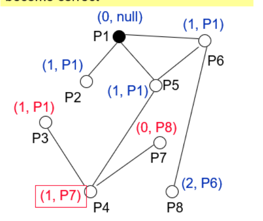
Properties. One cause of fault is the topology change from node movement. The algo is also periodical; we do not assume code is done atomically.

The protocol is not done round-by-round, but in "phase"s.

Self Stabilizing Spanning Tree Algorithm

Red values are initially incorrect values; blue values are values that have become correct

- On P1 (executed periodically):
  - dist = 0; parent = null;
- On all other processes (executed periodically):
  - Retrieve dist from all neighbors
  - Set my own dist = 1 + (the smallest dist received)
  - Set my own parent = my neighbor with the smallest dist (with tie breaking if needed)



Example execution. Say P_4 do their portion of code first; see neighbors and see 0,1,1 distance. They indeed update their value to a new tuple (BUT it's all still wrong). Then, P_4 can get lucky by favoring tiebreak in-favor/over P_7, rather than P_3.

Difference from leader election (the spanning-tree one). This one needs more rounds and get more information; and this algo needs to be correct in the presence of faults.

[Next lecture: correctness proof of algorithm.]

12 W12 was a holiday (Hari Raya Puasa), no lec

13 Correctness Proof of Self-Stabilization Algo

Phase. Define a phase to be "minimum time period where each proc has executed its code AT LEAST once" (i.e. technically, it's called "has taken an action") {some proc may execute its code more than once; the defn of a phase here is different from a round (in synchronous systems) (!)}.

Let A_i := len-of-shortest-path from proc i to root (called the LEVEL of proc i), and $dist_i$:= value-of-$dist$ on proc i.

Note: when topology not change, A_i is fixed and $dist_i$ may change.

Level properties: {a node at level X has at least one neighbor in level $X-1$; a node at level X can only have neighbors in level $X-1$, X and $X+1$ (and "nowhere" else)}.

Lemma. At end of phase 1 (and consequently, r — this is proven inductively!) $dist_{\text{root}} = 0$ and $dist_i \geq 1$ for any other i (consequently, any process i whose A_i is $\leq r-1$, we have $dist_i = A_i$; any process i whose $A_i \geq r$, we have $dist_i \geq r$).

Induction base already holds for $r == 1$; assume lemma holds at phase r, now consider phase $r+1$. By inductive hypo, know by end of phase r "some stuff(r)" must be true; we need to prove "some stuff($r+1$)" must be true (too).

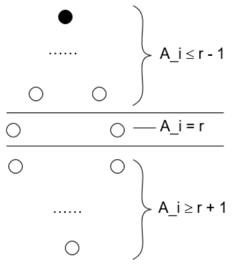
NOTE on proof trickiness. "Consider all t actions taken during phase $r+1$ (t may be >> r), use induction on t" this tech/proof-style is tricky because {a proc may take multiple actions in a phase (!); proc-s may take actions in parallel (i.e. interleaving-ly), can't assume a serialization of all actions (!)}.

The proof tech is typical for proving self-stabilization: {Step 1. (the t actions) will not roll back what alr achieved (so far) by phase r (i.e. no backward move); Step 2. at some point, each node will achieve more (i.e. forward move); Step 3. (any of the t actions) not roll back the effects of forward move after it (had) happened (i.e. no backward move after (the) forward move)}.

Claim 1: The blue conditions hold *throughout* phase r+1.

Proof: Use an induction on t. If t = 0, trivial. Next assume the statement holds for t and consider action (t+1) by some node i. (Cannot assume that action (t+1) happens after the t actions.)

for nodes with	already know: phase r	want to show: phase r+1
A_i ≤ r-1	dist_i = A_i	dist_i = A_i
A_i = r	dist_i ≥ r	dist_i = r
A_i ≥ r+1	dist_i ≥ r	dist_i ≥ r+1

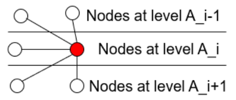


Proof (continued):

Case I: A_i ≤ r-1. Node i has at least one neighbor Y at level A_i-1. By the blue condition and by our inductive hypothesis, we know that Y's dist value is exactly A_i -1. Furthermore, node i only has neighbors at level A_i-1, A_i, and A_i+1. By the blue condition and 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 is A_i-1. Hence node i will set its dist value to be A_i.

Case II: A_i ≥ r. Similar…

for nodes with	already know: phase r	want to show: phase r+1
A_i ≤ r-1	dist_i = A_i	dist_i = A_i
A_i = r	dist_i ≥ r	dist_i = A_i
A_i ≥ r+1	dist_i ≥ r	dist_i ≥ r+1



Claim 2 + its proof. node i satisfy red conditions at some point (of time) during phase $r+1$. Proof: leverage on Claim 1's validity (!) (that blue conditions always hold throughout phase $r+1$

Claim 3 + its proof. for each node after (first) satisfies red condition, will continue to satisfy FOR THE REMAINDER of phase $r+1$. Proof: enumerate 3 cases + leverage that blue always holds (i.e. Claim 1).

Ending/proof-completion. Thm: after $H+1$ ($H :=$ length of shortest-path of most-far-away process to the root) phases, $dist_i = A_i$ on all processes, AND the parent values on all processes are correct.

Proof of first statement: trivial. Proof of second statement: by 1st statement (i.e. dist equals to its level) + fact that parent ptr points to $\underline{a}$ node of level $A_i - 1$. So each proc has path to root (implying graph being connected) and hence it is a spanning tree.