ass 21
6.852
Fall 01

big: chalk

1

Reading: 19, ~~[illegible]~~ 21 + Chadra - Fouez paper

Next: 22

Consistent Global Snapshots & Stable Property Detection

When Jim taught about logical time, (I think) he described ~~the application~~ of how logical time can be used to take a snapshot of a running distributed system.

Now examine the idea of taking snapshots more closely:

General idea is to monitor a given dist system A and determine some properties of A:

- Check for whether invariants are true.
- Check for termination, deadlock.
- Computing some function of the global state (e.g. total amt of money)
- Produce a complete snapshot for a backup.

Monitored version called B(A), transformed.

Not generally a formal composition of A with something — generally more closely coupled, allowing the monitoring process to look inside, at the state.

Clearly, Misra formalized the kinds of modifications that are permitted, ~~called~~ called modified versions superpositions.

Roughly speaking: can { add { new state components / new actions
modify old transitions, but only in certain "nonintrusive" ways

LTTR

2 main notions:

Consistent global snapshot
+
Stable property

Consistent Global Snapshot

Instantaneous snapshot = global state of system, procs + channels, at some point in an execution

CGS: not exactly an IS, but should "look to all procs" like an IS.

Useful for all the tasks listed above.

Stable property: A property of a global state s.t. if it ever becomes true, it stays true forever.
(E.g. termination, deadlock)

Connection: a CGS can be used to detect stable properties.

Termination detection:

a simple stable property detection problem

Dijkstra–Scholten:

assumes algorithm A starts with all nodes quiescent (only inputs enabled)

+ that input "start", trigger, arrives at only one node.

Doesn't need to be predetermined node.

From that node, computation can "diffuse" throughout network, or portion of network

(assume network is arbitrary connected undirected graph).

At some point, the entire system may become quiescent.
(no msgs in transit, no non-input actions of any nodes enabled).

Term. detection: If A ever reaches quiescent state then originating node should eventually output done.

TD ~~alg. [illegible]~~ is to be done by monitoring algorithm B(A), obtained by adding some stuff to each process of A:

D-S:

Augment A with extra pieces that construct + maintain a spanning tree of the currently active nodes – grows, shrinks, grows, ...
as nodes become involved, quiescent, involved again, ...

Tree rooted at source (node that gets input)

Informal description:

Add ack messages.

Ordinary msgs of A treated like search msgs in Asynch Spanning Tree

Each process accepts first incoming A msg as coming from parent.
acks other incoming msgs right away, but doesn't yet ack the parent's msg.

Source always acks immediately.

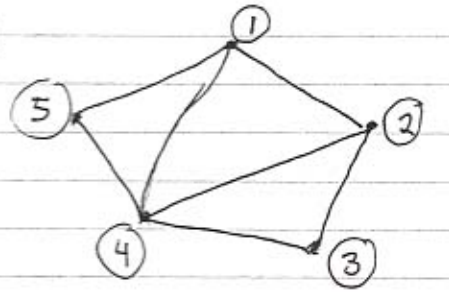
Allow tree to shrink using convergecast, to report termination back to source:

Each node looks for its own state to be quiescent, + all of its outgoing algorithm A msgs to be acked.
+ then cleans up locally: { acking its parent, + deleting all info about the termination protocol }

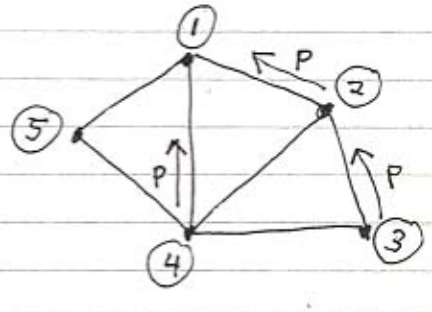
This cleanup means that, if it gets a new A msg later, it starts all over again

Thus, can grow, shrink, grow, ...

Ex:



① wakes up, sends msgs to ② + ④,
② sends to ③, establish parent pointers

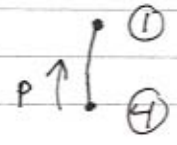


Next, ④ sends msg to ③ acked right away
④ " " " ① " " "

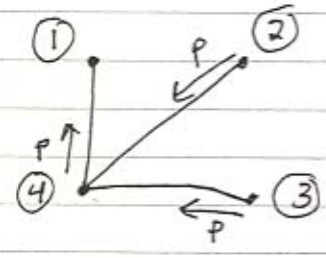
Now ① - ④ keep sending to each other for a while, everything acked immediately.
No one happens to send anything to ⑤.

Now ③ quiesces (locally)
② quiesces "
③ cleans up, acks ②, ② receives ack, ② cleans up

Thus, tree shrinks to just



Now at ④ sends msgs to ② + ③, get new tree:



Etc.

Go over code $\boxed{\text{p. 622-623}}$

Claim <u>correctly detects termination</u> (quiescence everywhere, no A msgs in transit)

Depends on some invariants, LTTR.

<u>Main ideas:</u>

<u>1. If B(A) announces termination then A is really done.</u>

Depends on invts saying { <u>if root is idle</u> then <u>so are all nodes</u> (not actively engaged in the termination protocol)
<u>if node is idle</u> then the part of underlying alg A at that node is <u>quiescent</u>.

<u>2. If A ever becomes quiescent then eventually B(A) announces termination.</u>

Pf: An inv says we always have a directed tree rooted at source, spanning all non-idle nodes (+ no others):

If A quiescent ~~[illegible]~~, this tree stabilizes (no new A msgs, others eventually finish).

What does it stabilize to? Claim it shrinks down to one node.
. If not, leaf node enabled to clean up ... contradicts stability of tree.

LTTR
more careful in book

$\boxtimes$

Complexity:
msgs: $2m$, where m is number from A
time: $O(md)$ from quiescence to done.
msg delay bound

These bounds are most interesting if $m << n$, ~~that~~.
This makes most sense for algs that don't involve the whole network, just a local portion.

Example: Recall Asynch BFS, with corrections. $\boxed{\text{Ch. 15}}$
Doesn't terminate, as I presented it earlier in the term.
But if we apply D-S to it, we get a terminating version.
~~Th~~ Ch. 15 discussed an ad hoc version of such a termination strategy — now you have seen it more systematically.

Consistent Global Snapshots

Given G connected, undirected.
A arbitrary S/R alg.
B(A) to take "snapshot".

Now any number (≥ 1) of the nodes may get $snap_i$ inputs, triggering the snapshot.

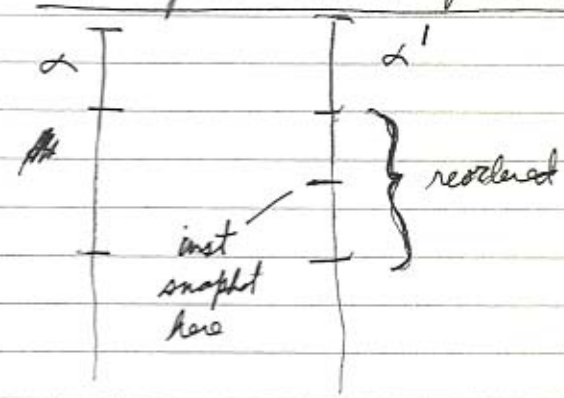
All nodes supposed to output $report_i$ containing $\begin{cases} \text{state for that node} \\ \text{states for all incoming channels.} \end{cases}$

Combination is a global state.

Want it to be such that:

If α is the embedded execution of A, then
$\exists \alpha'$, another " " ", such that:

1. $\alpha \overset{A_i}{\sim} \alpha'$ for each i
2. α' identical to α up to first snap + after first report
   (in between, same events, in possibly different order)
3. Reported state = actual global state after some intermediate step of α':



Thus, we're getting a consistent global snapshot, but only reordering a certain segment to get the instaneous snapshot.

Chandy–Lamport algorithm

Recall Logical Time Snapshot from last week
Get snapshot at given logical time. t.
Conceptually nice.
But depends on finding a nice value of t.
C-L can be viewed as a way of running the same algorithm, but without explicitly using any particular logical time.

Uses marker msgs to indicate where the logical time of interest occurs.
(Put marker between msgs with ltime (of sending event) $\leq t$ & ltime $> t$.)

Algorithm:
When uninvolved process receives snap$_i$:
snaps A_i's state
sends marker on each outgoing channel (marks boundary between msgs sent before + after the snap)
Thereafter, records all msgs arriving on each incoming channel, up to marker.

When process receives marker before receiving snap:
Then immediately records state of A_i (snap), sends out markers + begins recording incoming msgs as before.
Channel on which it got the marker is recorded as empty.

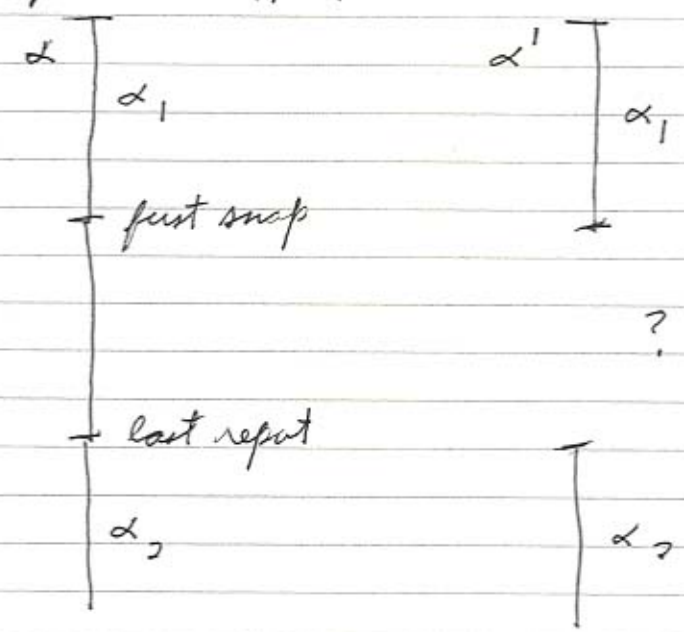
As over code, p. 628–629

Correctness:

First see that it terminates: Since all snap eventually and markers eventually sent + received on all channels.

Show it returns a correct state for some appropriate α':

Let α = the actual exec of A



Divide the events of A into S_1 and S_2

those after snap

those before the snap at their process

Note: all events belong to some process

Now reorder the middle group of events, putting all the S_1 events before all the S_2 events, while preserving {order at each process, send / receive order} = causality order

Can do this because: No send appears in S_2 while its receive appears in S_1.

This follows because of the marker discipline: the send in S_2

occurs after the snap, so after the marker is sent

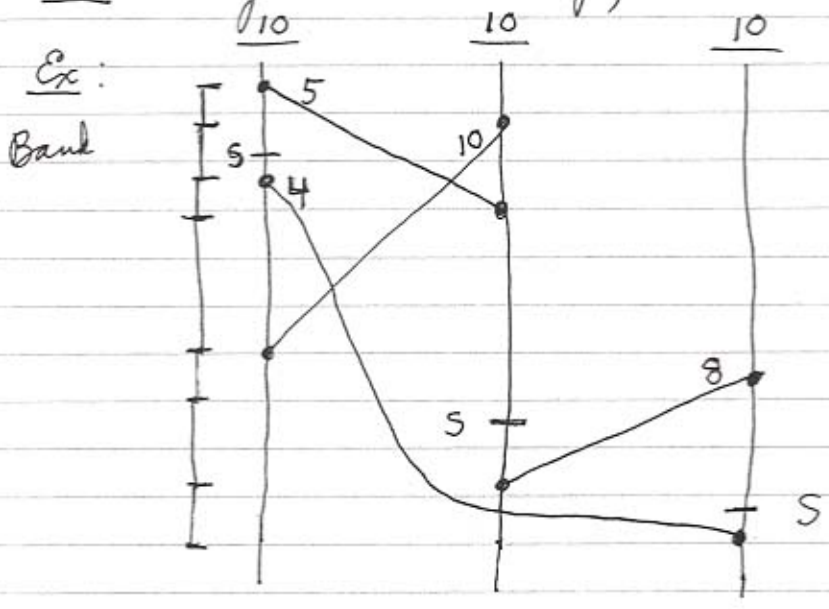
produces a msg that follows the marker on ~~all~~ its ~~outgoing~~ channels.
Recipient snaps when it receives the marker (if not sooner) + hence before receiving the msg. (so receive is in S_2)

So α' is the result of the reordering, + snap gives exactly the state after S_1.

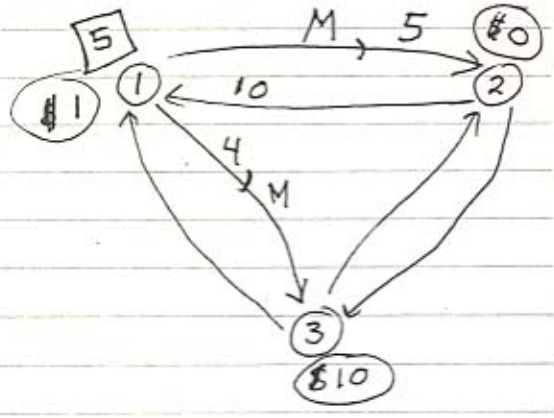
$\boxtimes$

Note: alg works in strongly-connected digraphs too.

Ex:



Suppose
1 sends \$5 to 2
2 sends \$10 to 1
1 snaps, gets $\boxed{5}$, sends markers
1 sends \$4 to 3



<u>Next</u> \$5 arrives at ②

\$10 arrives at ①, ① accumulates this in its count for channel (2, 1).

$\boxed{+10}$

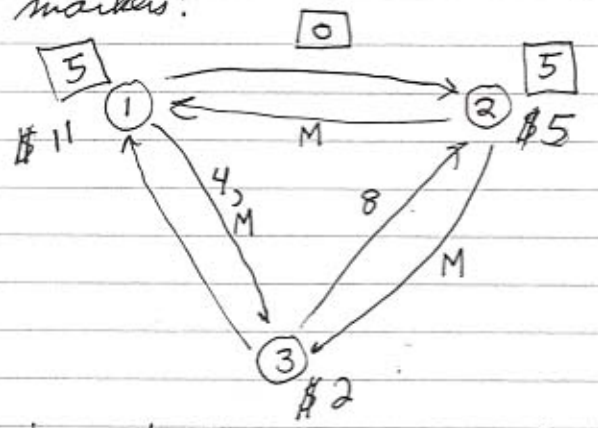
③ sends \$8 to ②

M in channel (1, 2) arrives at ②

② snaps, recording $\boxed{\$5}$ locally

+ $\boxed{\$0}$ for incoming channel (1, 2).

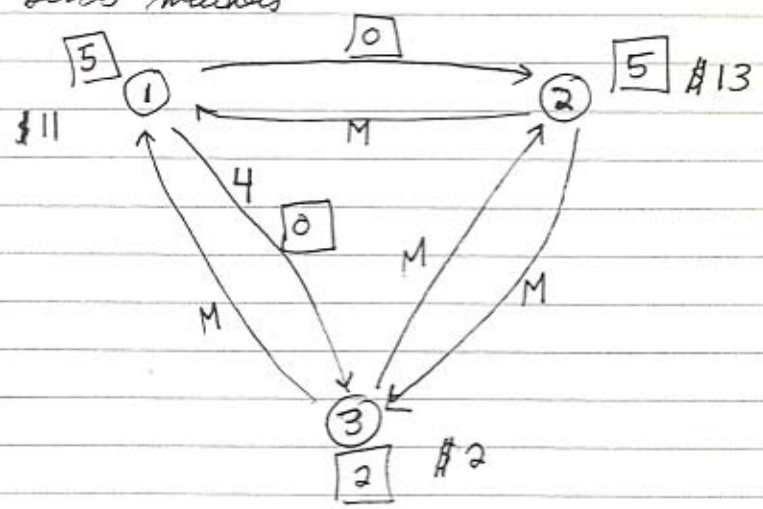
② sends markers:



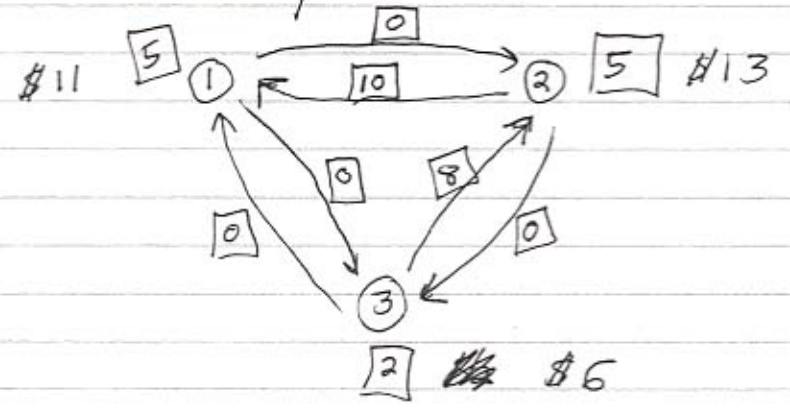
<u>Next</u>: \$8 arrives at ②, ② accumulates it in count for channel (3, 2)

M arrives at ③ from ①, ③ snaps, records $\boxed{2}$ locally, records $\boxed{0}$ for channel (1, 3)

sends markers



<u>Next</u>: 4 arrives at ③ (ignored for snapshot alg.)
Remaining Ms arrive, which closes the counts for the remaining channels.



<u>Check totals</u>: 30 = 10 + 10 + 10 (original split)
= 11 + 13 + 6 (cash at nodes now)
= (5 + 5 + 2) + (10 + 8) channels
nodes (snapshot totals)

Note this <u>snapped state never actually appears</u> in the real A execution. (check diagram)

But it <u>does appear in alternative execution</u> obtained by reordering events, lining up the local snapshots:

1 sends 5
2 sends 10
2 receives 5
3 sends 8
(snap ~~[illegible]~~ occurs here)

1 sends 4
1 receives 10
2 receives 8
3 receives 4

Complexity:

msgs: $O(|E|)$ – all edges, unlike DS

time: $O(\text{diam } d)$

(ignoring l, ignoring piggyb)

Applications:

Bank counting, as above.

Checking invariants:

- States returned are reachable global states, so an invariant should be true in such a state.
- Can check (before trying to prove)
- Checking requires some work:
  - Can collect the whole snapshot in one place + test the inv there.
  - Or, could keep the snapshot results distributed and ~~develop~~ use some kind of distributed algorithm to check the property.

  For some properties this is easy:
  - Local type, e.g., consistency of values at 2 ends of each edge

    send-ct(i,j) = rcv-ct(i,j) + # of msgs in transit on (i,j).

  Others harder: E.g. no cycles in a "waits-for" graph.

Stable Property detection

Similar to invariants, but they needn't be true in all reachable states.

SPs are Pathos, once true, stay true.

Ex: Termination

For algs without inputs, but where start states aren't (necessarily) quiescent.

Ex: Deadlock

We have a set of procs wait indefinitely for each other to release some resource.

(e.g. in dumb DP alg.)

Can use consistent global snapshot to detect (such) stable properties: Info this gives, for stable prop. P:

- If P is true of the snap state, then true in real state after final report. (+ thereafter)
- If P is false of snap state, then false in real state at beg., before first snap.

See these by a reachability argument.

So be sure of detecting, have to try periodically.

Ex: ~~Ter~~ Term. det. for BFS.

Just do Asynch BFS, changes repeatedly.

Do C-L term det periodically, to see if done.

Fault-Tolerant Network Computation

Rapidly running out of time this term, since last 4 classes will be devoted to presentations

Most of the remaining time will be spent on fault-tolerant communication.

But first, a brief look at Fault-Tolerant Network Computation, from Chapter 21
+ papers (in course bib) by { Chandra* + Toueg / Unreliable FD for Reliable Dist Systems
Hadzilacos + Toueg — various kinds of FT bcasts

Main topics: Failure detectors, consensus, + broadcast
Three problems to be considered for solution in asynch ~~distributed systems. network model.~~ distributed systems
Study separately, + relationships between them.

Starting point: Impossibility of consensus, tolerating even 1 failure, in asynch network.

But consensus is important.
Q: What can we add to the basic asynch network model that would make the problem solvable?
A: a failure detection capability, allowing procs to detect when other procs have failed.
Could be implemented, e.g., using timeouts, under certain timing assumptions.
in a real system

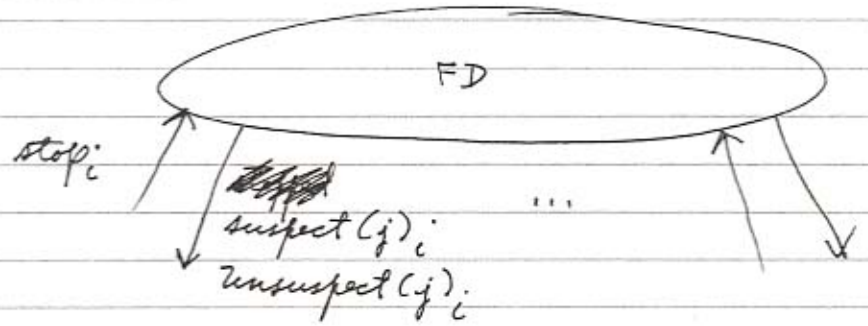
CT define several versions, varying in how well they do in:
completeness: detecting all failures
accuracy: avoiding mistaken suspicions
Show that even a fairly weak FD suffices to solve consensus.

Also define another problem, Atomic Bcast + show its equivalence with consensus. So impossibility + FD results apply to ABCAST also.

Model, defs in paper different from those we're using, so I'll try to re-cast in familiar terminology.

Consider process stopping failures only.

Failure detector:



Service with $stop_i$ inputs, gives $suspect(j)_i$ + $unsuspect(j)_i$ outputs, indicating latest opinion of whether j has failed.
Can change opinions
Opinions don't have to correspond to the actual failures.

Formally, an IOA.

FD properties:
Completeness: If $stop_i$ occurs + $stop_j$ doesn't, then eventually a final $suspect(i)_j$ occurs with no following $unsuspect(i)_j$.

(liveness property)

(They call this strong completeness, + also consider a weaker version, but I don't think it's interesting.)

Accuracy: Don't want to have too many suspicions. 4 interesting versions:

~~1. Perfect accuracy: P~~
~~If i never fails then no $suspect(j)_i$ ever occurs.~~

1. Perfect accuracy, P

If suspect(j) occurs then there's a preceding stop$_i$.

(only suspect ~~[illegible]~~ procs that have actually failed)

A perfect FD, with perfect accuracy + completeness, is considered in §21.4.

2. Strong accuracy, S

If every i is suspected by someone, then all procs fail.

(or, as they put it:
If not all procs fail, then some process is never suspected by anyone.)

3. Eventually perfect, $\Diamond P$:

There is some point in the execution after which no non-failed process is ever suspected by any non-failed process.

Allows some mistakes, but they have to stop eventually.

4. Eventually strong, $\Diamond S$

If not all procs fail, then there is some point after which some process is never suspected by any nonfaulty process.

$(\exists pt, \exists i)(\forall \text{ nonf } j)$ [j doesn't suspect i after that point].