Reading: 19, 21; Chodra - Foweg pope
Next: 22
When I wrote about logical time, (I think) I 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 distributed system $A$ and determine some properties of $A$:
  - Check for whether invariants are true.
  - Check for termination, deadlock.
  - Compute some function of the global state (e.g., total amount 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, Mean formalized the kinds of modifications that are permitted, called modified versions superpositions.

Roughly speaking: can add new state components

1. New actions
2. Modify old transitions, but only in certain "nonintrusive" ways

2 main notions:

- Consistent global snapshot
- Stable property
Consistent Global Snapshot

Instantaneous snapshot = global state of system, Proc + 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)

a) that input "shut", trigger, arrives at only one node.

Don't need to be predetermined node.

b) From that node, computation can "diffuse" throughout network, or
partition of network

(assume network is arbitrary connected undirected graph)

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

Termin. detection: if A ever reaches quiescent state, then originating
node should eventually output done.

To do this is to be done by monitoring algorithm B(A),

obtained by adding some stuff to each process of A:
Augment A with extra prices that construct + maintain a spanning tree of the currently active nodes - grow, shrinks, grow, ...
as nodes become involved, quieter, involved again, ...
See noted at source (node that got input)

Informal description
Add ack messages
Ordinary msg of A treated like search msg in Asynchronous Spanning Tree each process accepts first incoming A msg as coming from parent.
other incoming msgg right away, but doesn't yet ask the parent to msg.
Some always asks immediately
Allow tree to shrink using convergecast, to report termination back to source:
Each node looks for its own state to be quietest. + all of its outgoing msgg to be asked.
+ then cleans it locally: { asking its parent; + deleting all info about the termination protocol
This 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 msg to 2, 4
2 sends to 3, establish parent pointers
Next, 4 sends msg to 0 and 0 asks right away.

Now (1 - 4) keep sending to each other for a while, eventually asking immediately.
No one happens to send anything to 5.

Now 3 queries (locally)
   2 queries   
   6 cleans up, asks 2, 2 receives ask, 2 cleans up

Thus, tree shrinks to just:

At

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

\[ \text{Etc.} \]

p. 622-623

Claim correctly detects termination (quiescence everywhere, no A msgs)
in transit

Depends on some invariants, LTRR.

Main ideas:

1. If \( B(A) \) announces termination, then \( A \) is really done.

   Depends on invariants, if root is idle then so are all nodes.

2. If \( A \) ever becomes quickest, then eventually \( B(A) \) announces termination.
If an \\
int says we always have a directed tree rooted at source, \\
spanning all non-
idle nodes (+ no others).

If a\[\text{quiescent}\]

this tree stabilizes

(no new A node, others eventually finish).

What does it stabilize to? Claim it shrinks down to one node.

If not, leaf node enabled to clean up... contradict stability of tree.

LTTR

more careful in book

Complexity:

\[2^{m}\text{, where } m\text{ is number from A}\]

time: \[O(md)\text{ from quiescence to done.}\]

Other bounds are most interesting if \[m \ll n^{2}\].

This makes most sense for algo's that don't involve the whole network,
just a local portion.

Example: recall Augh BFS, with corrections. \[\text{Ch. 15}\]

Doesn't terminate, as I presented it earlier in the term.

But if we apply D-S to it, we get a terminating version.

Ch. 15 described an ad hoc version of such a termination strategy — now

you have seen it more systematically.
Consistent Global Snapshot

Given a connected, undirected.
A arbitrary S/R alg
\( \theta(A) \) to take "snapshot".

Now any number (\( \geq 1 \)) of the nodes may get snap inputs, triggering the snapshot.

All nodes supposed to output report containing state for that node
states for all incoming channels.

Combination is a global state.
Want it to be such that:

1. If \( \lambda \) is the embedded execution of \( A \) then
   \( \exists \lambda' \), another "", such that:
      - \( \lambda' \) identical to \( \lambda \) up to first swap + after first report
        (in between, some events, in possibly different order)
   - Returns state - actual global state after some intermediate step of \( \lambda' \):

Thus, we're getting a consistent global snapshot, but only reading a certain segment to get the instantaneous snapshot.
Chandy-Lamport algorithm

Recall logical time snapshot from last week.
Set 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 markers msgs to indicate when the logical time of interest
occurs.
(Put marker between msgs with $ltue$ (of sending event) $\leq t$)
+ $ltue > t$.

Algorithm:
When unindiced process receives snap:

- sends $A_{i}$ to state
- sends marker on each outgoing channel (marks boundary
  between msgs sent before and 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
  marker + begins recording incoming msgs as before.
  Channel on which it got the marker is recorded as empty.

For more info, p. 628-629
Correctness:

First see that it terminates. Since all snap eventually are marked eventually sent and received on all channels.

Show it returns a correct state for some appropriate \( \alpha \):

Let \( \alpha = \) the actual exec of \( A \). \( \alpha \)

\[ \alpha_1 \]

- first snap

\[ \alpha_2 \]

- last report

\[ \alpha \]

Divide the events of \( A \) into \( S_1 \) and \( S_2 \)

\[ \alpha \]

there after snap

there before the

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.

\[ = \text{causality order} \]

\[ \text{send} / \text{receive order} \]

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 \( S_2 \)
For any $i$, after the marker is sent, $x$ occurs after the marker. It follows the marker on all its outgoing channels. Recipient $n_{i+1}$ will, when it receives the marker (if not sooner), change its state by $x$ before receiving the msg. (so receive is in $s_2$).

So $x^1$ is the result of the reordering, and $\tau(n_{i+1})$ gives exactly the state after $s_1$.

Note: any node in strongly-connected digraphs too.

Ex: Bank

Suppose
1. sends $\$5$ to 2
2. sends $\$10$ to 1
1. snaps, gets $\$5$, sends markers
1. sends $\$4$ to 3
Next: $5$ arrives at $②$
$10$ arrives at $①$

$①$ accumulates this in its count for channel $(2, 1)$

$+10$

$②$ sends $⑧$ to $②$

$M$ in channel $(1, 2)$ arrives at $②$

$②$ swaps, records $⑧$ locally

$+⑰$ for incoming channel $(1, 2)$.

$③$ sends $⑪$:

Next: $⑧$ arrives at $②$

$②$ accumulates it in its count for channel $(3, 2)$

$M$ arrives at $②$ from $①$.

$③$ swaps, records $②$ locally

$②$ sends $⑪$.

$①$ records $⑦$ for channel $(1, 3)$.
Next: 4 arrives at 3 (ignored for snapshot alg.)

Remaining Ms arrive, which closes the counts for the remaining channels.

Check totals: \[ 30 = 10 + 10 + 10 \quad \text{(original split)} \]
\[ = 11 + 13 + 6 \quad \text{(each at mode now)} \]
\[ = (5 + 5 + 2) + (10 + 3) \quad \text{channels} \]
\[ \text{note, (snapshot total)} \]

Note this snapped state never actually appears in the real A execution (check diagram)

But it does appear in alternative execution obtained by reading events, lining up the snapshots:

1. sends 5
2. sends 10
2. receives 5
3. sends 8
(snap shot really occurs here)

1. sends 4
1. receives 10
2. receives 8
3. receives 4
Complexity:

msgs: \( O(1E1) \) — all edges, unlike DS

time: \( O(\text{diam } D) \)

(ignoring \( l \), ignoring helpers)

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 next time.
  - Or, could keep the snapshot results distributed and develop use some kind of distributed algorithm to check the property.

3 or some properties this is easy:

- Local type, e.g.: consisting of values at 2 ends of each edge

\[
\text{send-ct}(i,j) = \text{recv-ct}(i,j) + \#	ext{ of msgs in transit on } (i,j).
\]

Other kinds: E.g. no cycles in a "wait-for" graph.

Static property detection

Similar to invariants, but they needn't be true in all reachable states, rather, once true, stay true.
Ex: Termination

For algo without input, but where start states aren't (necessarily) quiescent.

Ex: Deadlock

Write a set of peers 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 queue, for stable prop. P:
- If P is true of the snap state, then true in real state after final report. (thenafter)
- If P is false of snap state, then false in real state at beg., before first snap.

See these by a probability argument.

To be sure of detecting, have to try periodically.

Ex: Sem. det. for BFS

Just do regular BFS, check repeatedly.

Do c-l term det periodically, to see if done.
Rapidly running out of time this term, nine 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 Natural Computation,
from Chapter 21

Unreliable FD for
+ papers (in course Oct) by Schneider, Tonge, Reliable Distributed Systems
Hadjieleous, Tonge

Main topics: Failure detection, consensus, broadcast.

Three problems to be considered for solution in asynchronous distributed
network models: distributed systems
study separately, + relationships between them.

Starting point: Impossibility of consensus, tolerating even 1 failing
in asynchronous network.

But consensus is important.

A: What can we add to the basic asynchronous network model that
would make the problem solvable?

A: A failure detection capability, allowing processes to detect when
other processes have failed.

Could be implemented, e.g., using timeouts, under certain timing
assumptions.

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 Broadcast + show its equivalence
with consensus. So impossibility + FD results apply to ABCast also.
Model, def in paper different from those we're using, so I'll try to re-cast in familiar terminology.
Consider process stopping failing only.
*Failure detector:

![Diagram]

Serve with stop; inputs, quick suspect (j); i: unsuspect (j).
Output, 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; occurs, stop; doesn't, then eventually a final suspect (i) occurs with no following unsuspect (i).

(\textit{likeness property})

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

Accuracy: Don't want to have too many suspicions. I'm interested in:

1. \textit{Strict accuracy:}
   If j is most likely to be suspect (j), i: unsuspect.
1. Perfect accuracy, $P$
   If suspect $(i)$ occurs then there's a preceding stop.

   (only suspect processes 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 processes fail.

   (or, as they put it:
   If not all processes 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 processes fail, then there is some point after which some process is never suspected by any non-failed process.

   $(\exists p; \exists i; (\forall non-f)y) \quad [i$ doesn't suspect $i$

   after that point $]$