Reading: 19, 21 + Charles - Foreign paper
Next: 22
Consistent Global Snapshots + Stable Property Detection

When you 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.
- 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.

Clancy, Misra formalized the kinds of modifications that are permitted, called modified version superpositions.

Roughly speaking: can add new state components:
- New actions
- Modify old transactions, but only in certain "nonintrusive" ways

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 – Schütte**

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 inputs 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 is to be done by monitoring algorithm B(A), obtained by adding some stuff to each process of A.
Augment A with extra pieces that construct + maintain a spanning tree
of the currently active nodes: grow, shrinks, grows, ...
as nodes become involved, quiescent, involved again, ...

Tree rooted at source (node that gets input)

Informal description:

Add node messages:
Ordinary msgs of A treated like search msgs in Approx Spanning Tree.
Each process accepts first incoming A msg as coming from parent.
All other incoming msgs right away, but doesn't get ack
the parent's 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 quiescent, + all of its
outgoing msgs to be asked.
+ then cleans up locally:
  asking 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  2
  \  /
  4  3
```

1 makes up, sends msgs to 2 & 4,
3

2 sends to 3, establish parent pointers
Next, 4 sends msg to 3 asked right away

Now 1 - 4 keep sending to each other for a while, everything asked immediately.

No one happens to send anything to 5.

Now 3 quenches (locally)

2 quenches

3 cleans up, asks 2, 2 receives ask, 3 cleans up

Thus, tree shrinks to just:

At

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

Etc.

Go even code [p. 622-623]

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

Depends on some invariants, LTTR.

Main ideas:

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

Depends on invariants, root is idle then so are all nodes, that is not actively engaged in the termination. Indeed the node is idle, part of underlying alg A at that node is quiescent.

2. If A ever becomes quiescent then eventually B(A) announces termination.
Pf: An innt says we always have a directed tree rooted at source, spanning all non-idle nodes (+ no others).

If a quiescent tree, this tree stabilizes
(no new A rings, others eventually finish).

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

If n'ty leaf node enabled to clean up... contradicts stability of tree.

LTTR
more careful in book

Complexity:

maps: \(2^m\), where \(m\) is number from A
time: \(O(md)\) from quiescence to done.

They sounds are most interesting if \(m \ll n\).

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

Example: recall anych BFS, with corrections. [Ch. 15]

Doesn't terminate, as I presented it earlier in the text.
But if we apply D-S to it, we get a terminating version.

Ch. 15 discussed an ad hoc version of such a termination strategy — now
you have seen it more systematically.
Consistent Global Snapshots

Given a connected, undirected, arbitrary S/R alg.
6(a) To take "snapshot''.
Now any number (≥1) of the nodes may get "snap inputs'', triggering the snapshot.

All nodes supposed to output report, containing state for that node's state for all incoming channels.

Combination is a global state.
Want it to be such that:
If \( x \) is the embedded execution of \( \lambda \), then
\[ \exists x', \quad x' = x \quad \text{such that:} \]
1. \( x \) \( \sim \) \( x' \) for each \( i \)
2. \( x' \) identical to \( x \) up to first snap + after first report
   (in between, some events in possibly different order)
3. Retrieved state = actual global state after some intermediate step of \( x' \):
\[
\begin{align*}
\text{step 1} & \quad x \quad \xrightarrow{x'} \quad x' \\
\text{step 2} & \quad \text{inst.} \\
\text{inst. report}\end{align*}
\]

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.

Let 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 \( \text{line} (\text{of sending event}) \leq t \)

+ \( \text{line} > t \)).

Algorithm:

When uninvoiced process receives snap:

- \( \text{snaps} A_i \) to 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

- marker + begins recording incoming msgs as before.

- Channel on which it got the marker is recorded as empty.

Go to code, p. 628-629
Correctness:

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

Show it returns a correct state for some appropriate $x^1$:

Let $x$ be the actual exec of $A$

\[ x \]

\[ x^1 \]

\[ x^2 \]

- first snap

- last report

Divide the events of $A$ into $S_1$ and $S_2$

\[ S_1 \]

\[ S_2 \]

- 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

$\text{send} / \text{receive order}$

$= \text{causality order}$

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 $S_2$
Suppose 1 sends $5 to 2
2 sends $10 to 1
1 snaps, gets 5, sends markers
1 sends $4 to 3
$5$ arrives at $2$
$10$ arrives at $1$ → $1$ accumulates this in its count for channel $(2,1)$

$3$ sends $8$ to $2$

M in channel $(1,2)$ arrives at $3$

$2$ snaps, records $5$ locally

+ $80$ for incoming channel $(1,2)$.

1. Sends markers:

\[ \begin{array}{c}
\text{1} \\
\text{M} \\
\text{M} \\
\text{M} \\
\text{2} \\
\end{array} \]

$8$ arrives at $2$, $3$ accumulates it in count for channel $(3,2)$

M arrives at $2$ from $1$, $3$ snap, records $2$ locally

records $0$ for channel $(1,3)$

1. Sends markers:

\[ \begin{array}{c}
\text{1} \\
\text{M} \\
\text{M} \\
\text{M} \\
\text{2} \\
\end{array} \]
Next: 4 arrives at (2) (ignored for snapshot alg.)

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

Check totals: $30 = 10 + 10 + 10$ (original split)
               $= 11 + 13 + 6$ (cash at nodes now)
               $= (5 + 5 + 2) + (10 + 8)$ (channels, snapshot totals)

Note this snapshot 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 snapshot everything occurs here)
1 sends 4
1 receives 10
2 receives 8
3 receives 4
Complexity:
msg: \( O(1E) \) — all edges, unlike DS
time: \( O(\text{diam } d) \)
(ignoring \( l \), ignoring policies)

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 unit there.
  - Or, could keep the snapshot results distributed and develop use some kind of distributed algorithm to check the property.
  - On some properties, this is easy:
    - Local type, e.g., consisting of values at 2 ends of each edge
    \[
    \text{send-ct}(i,j) = \text{rev-ct}(i,j) + \text{# of msgs in transit on } (i,j). \]

Others hard:
E.g., no cycles in a "wait-for" graph.

Stable Property Detection
Similar to invariants, but they needn't be true in all reachable
SPs are states.
Pathwise, once true, stay true.
Ex: Termination
For algos without inputs, but whose start states aren't (necessarily)
quiexent.

Ex: Deadlock
Where a set of proc 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 quiz, 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 as quick BFS, changes repeatedly.
Do C-L term det. periodically, to see if done.
Fault-Tolerant Network Computation

Rather 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

Main topics: Fault detection, consensus, & broadcast

Three problems to be considered: for solution in any distributed system. Stated separately, & relationships between them.

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

But consensus is important.

A: What can we add to the basic asynchronous 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.

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 recast in familiar terminology.

Consider process stopping failures only.

Failure detector:

\[
\begin{array}{c}
\text{FD} \\
\text{stop}_i \\
\text{suspect}_i (j_i) \\
\text{unsuspect}_i (j_i) \\
\ldots
\end{array}
\]

Service with \(\text{stop}_i\) inputs, gives \(\text{suspect}_i (j_i)\) or \(\text{unsuspect}_i (j_i)\) outputs, indicating latest opinion of whether \(j_i\) has failed.

Can change opinions.

Opinions don’t have to correspond to the actual failures.

Formally, an IoA.

\textbf{FD properties:}

\textbf{Completeness:} If \(\text{stop}_i\) occurs \& \(\text{stop}_j\) doesn’t, then eventually a final \(\text{suspect}_i (j_i)\) occurs with no following \(\text{unsuspect}_i (j_i)\).

\textit{(liveness property)}

\textit{(They call this strong completeness, \& also consider a weaker version, but I don’t think it’s interesting.)}

\textbf{Accuracy:} Don’t want to have too many suspicions.

\textbf{Version:}

1. Perfect accordance.
2. Better than no suspect \(j_i\), even worse.
1. Perfect accuracy, $P$

If suspect $i$ occurs then there's a preceding step.

(only suspect processes that have actually failed)

A perfect FD, with perfect accuracy and 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, $◊P$:

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, $◊S$

If not all processes fail, then there is some point after which some process is never suspected by any non-failed process.

$(\exists p, \exists i) (\forall \text{non-f} j [j \text{ doesn't suspect } i \text{ after that point}])$