EE 382N: Distributed Systems **Fall 2017**

Lecture 2: Sept 16

Lecturer: Vijay Garg *Scribe: Robert Pate*

2.1 Introduction

So far this course has covered clocks, resource allocation with distributed locks, and taking system snapshots. Next we use all of them to build a sensor that can check for some global property. We'll be able to answer questions like "did my system go through a state where property B was true?" and "is my system in deadlock"?

2.2 Stable Properties - Deadlock

Definition 2.1 *A predicate B is "stable" if once it becomes true it stays true, e.g. deadlock and loss of a token.*

Claim 2.2 *We have a program to detect a stable predicate*

How? We will use our established camera/snapshot algorithm to take snapshots of a program. Build a wait-for graph using everyone's state and check if it is cyclic.

Consider a system S where we initiate a snapshot at S_{i} and it's finished at S_{f} giving us S^* as the snapshot. We'll apply S^* to $B()$ which returns true if deadlock.

Two cases: $B(S^*)$ shows deadlock or it does not.

$B(S^*) \rightarrow B(S_f)$
S_f is reachable from $B(S^*)$ so it has deadlock

if $\neg B(S^*) \rightarrow \neg B(S_i)$
There was no deadlock at S_i but it may still happen so we should take more snapshots.

2.3 Unstable Properties

Let's say $B \equiv (P_1 \text{ in CS}) \wedge (P_2 \text{ in CS})$
(Two processes are in the critical section at once.)

If we take a bunch of snapshots and check B for each then we still won't know if B was ever true. How can we detect this kind of property? When should I take the picture?

Claim 2.3 *If we store the history of each process, we should be able to find one state from each process where we can apply B to find it true.*

Assume m states per process and n process so the possible states is m^n. This is "combinatorial explosion" and is "not good." But it is at least finite!

Can we do better? Let's define the problem clearly.

Definition 2.4 *Global Monitoring Problem / Predicate Detection*

Does there exist G such that
(G is a Consistent Global State) $\wedge B(G)$*?*

Definition 2.5 *Consistent Global State*

$\forall i, j : G[i] \parallel G[j]$

2.3.1 Simplifying Assumptions

1. We have vector clocks since we must have them to determine concurrency in "perfect isomorphism" ($\parallel$)
2. B is conjunctive, meaning it's of type $l_1 \wedge l_2 \wedge l_3 \ldots \wedge l_n$
   - Thus we can rewrite B to put the local predicate(s) of each process first
   - e.g. if l_1 and l_3 are local to P_1 then we rewrite B to be $l_1 \wedge l_3 \ldots \wedge l_n$
   - And we can combine multiple predicates local to a process

Definition 2.6 *Given a conjunctive B, does there exist a G such that:*

(G is a Consistent Global State) $\wedge \forall_i : l_i[G[i]]$

So we find one state from each process. If its local predicate is false then the rest of the predicate must be false and we can ignore that state. We could label his "red" in a diagram.

So say we have a centralized algorithm, it could ignore all the "red" states and only start with the first global state that doesn't have a false predicate in it.

Next we start moving through these states to see if they're actually a "consistent global state." If any events in the happened before the other, it's not consistent. So advance one state on that process and check it for consistency.

Eventually either:

- A: we find our consistent global state where $B()$ is true OR
- B: we get to the end of the lane and $\neg B()$

2.4 Algorithm for nonstable conjunctive predicate detection

- have a centralized checker process
- keep a queue of vector clocks from each process

- find vectors that are *pairwise concurrent* and the predicate is true

Definition 2.7 *Pairwise concurrent = part of a consistent global state in the state based model*

Each process, P_i, whenever the local predicate is true, will send the vector clock to the *checkerprocess*.

Think of colors for vectors, e.g. red means you're less and green you're not. (note red here is going to mean a different thing than the red mentioned earlier).

Checker Process steps: $varnqueues; initwiththevectorsallpaintedred$

while there exists a queue with a red vector at the head of the queue:

1. throw away that vector
2. paint the next vector, v, green
3. compare v to any green vectors at the top of the other queues
   (a) if v is less than any of them, paint it red
   (b) if there's no next vector, end
   (c) if v is greater than some green vector u, paint u red

The goal here is to get everyone concurrent, and anyone less than another means it happened before and can be thrown out.

We want to wait for the next vector of a process that's still running.

2.4.1 Complexity

Message Complexity: overhead for sending vectors is $O(mn)messages$

Time Complexity: $O(mn)$ loops, $O(mn^2)$ time to make green, $O(n)$ time to compare vectors.

Space Complexity: $O(mn^2)$ integers.

2.5 Weak Conjunctive Predicate Detection Algorithm (WCP)

We can make this more practical by sending less vector clocks.

1. Currently we send a vector clock each time the predicate is true
2. Assume no messages were sent or received since the last time we sent our clock
3. Then the clock would not have changed!
4. So we should not send the same clock multiple times.

For process P_i :
Whenever local predicate becomes true for the first time in any *message interval*, send the vector to the checker process.

(*Message interval* means the time between messages to or from other processes.)

2.5.1 Can we improve on WCP?

Can we improve this to handle ORs?

We can convert any boolean expression to the *Conjunctive Normal Form (CNF)* by distributing the ORs into separate predicates.

Example:
$B \equiv (x = 3) \wedge (y = 5) \vee (z = 5))$ converts to:
Predicate 1: $(x = 3) \wedge (y = 5)$ Predicate 2: $(x = 3) \wedge (z = 5)$

Furthermore any negation can be pushed inside the predicates so that we only have $\wedge$ and $\vee$ left which can be converted to CNF.

Now we can run the algorithm for each predicate, but we've glossed over the real problem. The conversion to CNF can produce huge lists of predicates!

2.5.2 NP Complete Proof

Theorem:
Given a boolean expression B and a computation $(S, \rightarrow)$, the problem of detecting $\exists$ consistent global state of $(S, \rightarrow)$ *s.t.* $B(G)$ is NP-Complete (even when it's not distributed across processes).

Proof that PRED is harder than SAT:
If you're trying to find a consistent cut where a predicate is true, that's the same as the Boolean Satisfiability Problem (SAT) where you try to find a satisfying assignment for a boolean formula. SAT has been proved NP-Complete and therefore this is too.

2.6 Distributed / Token Based Conjunctive Predicate Detection Algorithm

For the distributed version, we won't use a checker process. Each process will have its own vector queue. We still need one process to check the queue, but we can distributed that task with a token.

The given checker process only needs the head of the queue so we can bundle that with the token.

Assumptions:

- We'll call pairwise concurrent vectors green
- We'll call vectors that happened before red
- Mattern vector clocks are in use
- Every vector is local snapshot when the predicate was true

Consider the matrix of vectors we had when centralized:

pid status	*vector clock queue*
$p1$ G	$1...n$
$p2$ R	$1...n$
$p3$ G	$1...n$

If these were all green then we're done. But we're not so we need to advance p_2.

In the distributed version, we don't need every full vector from every process. We'll have our own and the head of the others.

pid status	*head of vector clock queue*
$p1$ G	.
$p2$ R	.
$p3$ G	.

The process with the token advances until it finds a green in its queue. If there are other reds, it sends the token to any one of them. If there are no reds, then we're done.

We can make this yet more efficient since we're trying to answer the question "Is my value the biggest in my column?"

pid color	$p1clock$	$p2clock$	$p3clock$
$p1$ G	4	4	12
$p2$ R	7	9	15
$p3$ G	3	7	11

So we improve by only storing:

1. diagonal of the matrix (our local clock values)
2. max value from each column of the diagonal (max anyone knows)
3. color of each

Complexity:

number of token messages: $O(mn)$ where n processes and m worst case vectors

message size: $O(n)$ (this is a big improvement over $O(n^2)$

space: $O(m.n)$

Every time a token message is received, one vector is removed.

Further improvement: just keep the max from each column. I only need to know if i am that max or not which is already the color.