1 Lecture 3 Consistency Models

1.1 Definitions

1.1.1 History

The set of all events (invocation and responses) ordered by wall clock. Histories are always totally ordered.

1.1.2 Equivalent histories

Two histories are equivalent if they contain the exact same events.

1.1.3 Sequential history

A history is sequential if every *response* follows its *invocation* immediately with no interleaving.

1.1.4 Concurrent history

A history that has some interleaving.

1.1.5 Legal sequential history

A sequential history is *legal* if the *sequential semantics* of the data types are satisfied.

1.1.6 External order

Is the relation that e1 < e2 if e1's response happens before e2's invocation.

1.2 Sequential consistency

A history is *sequentially consistent* if it is *equivalent* to some *legal sequential history* that preserves *process order*.

- Intuitively, break history into processes, and feel free to slide the (inv, resp) pairs in a process until there is an execution which all objects behave as expected.

1.3 Linearizability

A history is linearizable if

- Definition 1: history is equivalent to some other history where each (inv, resp) pair is treated as a point between them.
- Definition 2: there exist an equivalent history that is legal and preserves the external order (which implies process order).

1.3.1 Linearizability is a local property

A history H is linearizable $\iff$ H|x is linearizable for all objects x.

1.3.2 Linearizability implies sequential consistency.

Easily seen in definition 2: external order is a superset of process order.

2 Lecture 4 Models and Clocks

2.1 Logical Clock

- form: a nonnegative integer, initialized to 0
- increment on process: += 1
- sending: just my clock
- upon receive from other process: my_{clock} = max(my_{clock}, $their_{clock}$) + 1

2.2 Vector Clock

- form: a vector of `n` nonnegative integer, initialized to all 0
- increment on process: if my index is i, v[i] += 1
- sending: just my clock
- upon receive from other process: my_{clock}[k] = max(my_{clock}[k], $their_{clock}$[k]) for all k

2.3 Matrix clock

- form: a `n x n` matrix of nonnegative integer, initialized to all 0
- intuition: the ith row is what i know.
- increment on process: if my index is i, v[i][i] += 1
- sending: just my clock
- upon receive from other process: update their row. update my row as per vector clock - pairwise max.

3 Arrows

- $a \prec b$ is process order.
- $a \to b$ is happen-before order: the logical clock value is less than, iff there is a directed path. Clearly $a \prec b \implies a \to b$.
- $a \rightsquigarrow b$ is the send-receive order.

4 Lecture 5 Global Snapshot

Assumption: channels are FIFO. If not, it is discussed in Message Ordering lecture

4.1 Definitions

4.1.1 Global snapshot

A set of events `E` such that for any event `e` in `E`, if $f \prec e$, then $f \in E$.

4.1.2 Consistent global snapshot

A global snapshot where if e_1 send, e_2 receive, e_2 in then e_1 must be in.

4.2 Chandy & Lamport's protocol for taking snapshots

- Key idea:

1. after a process takes a snapshot, it orders others to stop via a message
2. If they already took a snapshot before the order arrives, then capture all message till the order (these are the on-the-fly messages)
3. Otherwise once they receive the order, stop. (otherwise will receive stuff which sender does not include)

4.3 Theorems

- On any process, given a positive integer M, there exist a global snapshot containing all events up to and including M, and does not include anything after.

5 Lecture 6 Message Ordering

We still assume FIFO (except for fully async order) for inter-process channels.

5.1 Definitions

5.1.1 Fully Asynchronous

No restriction.

5.1.2 FIFO

Messages s_1, s_2 are sent from i to j. Then $\neg(s_2 \prec s_1)$.

5.1.3 Causally Ordered

- Motivation: if s_1 caused s_2, then we must ensure r_1 is before r_2 if the receives are on the same process.
- Causal relationship is not modelled. However, more broadly, if s_1 happened before s_2, it could cause s_2.
- If s_1 happen before s_2, r_1, r_2 on the same process, then $r_{1\prec}

r_2$.

5.1.4 Synchronous Ordered

Diagram can be redrawn into vertical, respecting happen-before, process and send-receive order.

5.1.5 Totally ordered broadcast

If on p1, \(x

5.2 Algorithm to ensure Causal Ordering

5.2.1 Core Idea

When a process sends a message, attach also every message you have seen or sent so far.

5.2.2 Why does it work?

- Recall causal ordering: if s1 happens before s2 (there is a directed path), then r1 must before r2
- If s1 and s2 are on the same process, then attaching all messages it has sent will include s1 for the target process to see before s2
- If not on the same process, then attaching all messages it has received will include s1 for the target process to see before s2.

5.2.3 Can we improve it?

- Just include messages that are targeted at the receiving process.
- Or number each message locally on the process. Instead of attaching all messages, make the target wait until all processes' intended message to it has been delivered.

5.3 Skeen's Algorithm for Broadcast Ordering

- An example is a chat group, sending messages to everyone
- Everyone needs the same message number for the same msg

5.3.1 Core Idea

1. First round - send message to everyone to save in the buffer. After saving, each process replies the coordinator their current logical clock number.
2. Wait (blocking) for all them to come back, and assign the message the largest logical clock number.
3. Let everyone know the decided message number.
4. Continue as per usual.

5.3.2 Summary

It orders messages, not logical clock numbers. Each logical clock value run as usual.

6 Lecture 7 Leader Election

6.1 Motivation

Leader election can

1. Coordinate messages. It trivially solves totally ordered broadcast by having the leader numbering the messages.
2. Control over shared memory, trivially solving mutual exclusion.

6.2 Node Topologies

Different arrangement of nodes give rise to different algorithms.

6.2.1 Rings

1. Anonymous rings - impossible for deterministic algos
    1. Known ring size - randomized algorithm
        1. Every node choose a random number from 1 to n, then run Chang-Roberts algorithm.
        2. Every node attach a $read_{count}$ tag on each message. If a node receive its id with $read_{count}$ = ring size, it is (one of the) winners.
        3. Winners go to a second round. If the links are FIFO, it will know it is the leader if no message was received before getting own message.

1. Analysis - ensure termination

   Idea: call a round good if it kicks out a node. We denote the number of rounds between good round i and i+i as x_i, and deduce that x_i drops below $r/(n-1)$ as r, the number of total rounds get huge. With a union bound, we can show that the total number of rounds is less than r goes with probability 1.

2. Unknown ring size - impossible

   Consider two rings, one with one node, the other with two. Then a node cannot distinguish it is in the first kind or the second kind, so any algorithm that solves will declare both processes as leader in the second ring.

2. Numbered rings - Chang-Roberts Algorithm
   1. Setting

      Given a ring of nodes that can only send messages clockwise, select a leader

   2. Algorithm
      1. Sending: Every node send its number clockwise
      2. Receiving: Every node relay the message (clockwise) if the value is bigger than self.
      3. If receive own id, then it is the leader
   3. Complexity
      1. Message Complexity

         Number of messages sent

      2. Best Case = O(n)

         Condition: sorted in clockwise ascending 1+…+1+n = 2n-1

      3. Worst case = O(n^2)

         Condition: sorted in clockwise descending 1 + 2 + … + n = n(n+1)/2

      4. Average case = O(nlogn)
         1. Consider the random variables x_k denoting the number of messages caused by node `k`. We want to find the expectation of the sum of all x_k, and by linearity of expectation, it is the sum of the expectation of each x_k.
         2. Fix `k`.
            1. For $x_k = 1$, we must have the node clockwise of `k` be greater than `k`. The probability is $\frac{n-k}{n-1}$.
            2. Repeat for other values of x_k. We will get $P[x_k = i | x_k > i-1] = \frac{n-k}{n-i}$ by similar analysis.
            3. Note that all these values are at least $p = \frac{n-k}{n-1}$.
         3. Consider a lottery where each ticket has a probability of p to win. The expectation of number of tickets to win is $1/p$. Since we have probabilities of at least p for each number of tickets (corresponsdingly number of messages), the expectation $E[x_k] \geq E[y] = 1/p = \frac{n-1}{n-k}$.
         4. Summing up through $k = 1, \ldots, n$, and using the fact that $1 + 1/2 + \ldots + 1/n = O(\log n)$, we have $O(n \log n)$ as desired.

6.2.2 Complete graphs

1. Know how many nodes there are in the network (protocol below).
2. Trivially broadcast own id to every neighbour (all n-1). When receiving an ID,

reply with the samd ID if it is larger. Node that receives n-1 replies is the leader.

6.2.3 Any connected graph

1. Construct a spanning tree.
    1. Initiating node (can be anyone) sends out "child requests".
    2. Any node receiving "child request" respond to one of them. Recursively send out "child requests".
    3. You are the root of the node if you don't have a parent. And those children that respond to you are your children.
2. Now with the spanning tree we can aggregate information easily.
    1. Eg. counting total # of nodes. Root node ask children to recursively count size, and just sum them up. Leaves respond 1.

7 Lecture 8 Consensus

7.1 Timing Models

7.1.1 Synchronous

- Bounded amount of time to do processing (generate output msg, process input msg)
- and bounded amount of time to send messages

7.2 Goals/Conditions for consensus

7.2.1 Termination

7.2.2 Agreement

7.2.3 Validity

7.3 Version 0: No failure (regardless of timing model)

7.3.1 Algorithm

For any process:

1. Keep broadcasting own value
2. Once confident have all messages (due to known ub), run a deterministic algorithm on all values (eg. max/min)

7.3.2 When does it work?

1. TODO Synchronous
2. TODO Async

7.4 Version 1: (Synchronous) Node crash failures

7.4.1 Setup

- Synchronous, all nodes have bounded processing time, all message has bounded delay
- Crash failure - crashes forever. On a "round" that it crashes, broadcast might not happen/not be complete.

7.4.2 Intuition

- We use rounds to gain information on crash: if no reply, then crashed.
- How to delineate rounds: let t1 be delay to generate output, t2 for message propagation, t3 to process input, then each round is simply `t1+t2+t3` long.

1. Suppose we don't have accurate clock
    - Claim: every clock should be some constant multiple of other clocks.
    - Consider a case where clock = 2x accurate clock.
        - Attempt 1: everyone set round duration to `2(t1+t2+t3)`. The process with faster clock will advance rounds faster. Then every process that receives a new message starts a new round. But what if a round starts before a process has time to process its received message?
        - Attempt 2: We provide the offset, making the round duration `2(t1+t2+t3+(t1+t2))`, the new offset tolerates the message delay(process output, send msg) to the "latest" node.

7.4.3 Protocol

1. Intuition
    - suppose there is no failure, we can just broadcast our message to everyone and conclude.
    - Since there can be a failure, we can **"forward"** all messages received in our broadcast.
    - What is left is to decide how many rounds are needed.
2. Details
    1. Intialize the set `S:={my_input}`.
    2. Broadcast S to everyone for `f+1` rounds (where f is number of node crashes to tolerate)
    3. Union S with all the sets received in this round, and repeat 2.
3. Correctness
    1. Termination is obvious (f+1 rounds, bounded waiting)
    2. Validity is obvious (if everyone same input, S = {s} is singleton)
    3. Agreement
        1. Given f+1 rounds and f failure, there must be at least 1 round with no failure. Call that the good round.
        2. See that the good round must end with all nodes (that haven't crashed) with the same set `S`.
        3. After the good round, each nodes's `S` do not change anymore.
        4. Then the deterministic function will choose the same value for all surviving nodes.

7.4.4 Lower bound is (Omega(f))

- Statement: any deterministic algorithm that works (fulfills termination, agreement, validity) must take at least f+1 rounds.
- Proof is too hard (not in scope).

7.5 Version 2: (Synchronous) Link Failure

7.5.1 Setup

- nodes do not fail
- but the message channels (between any pair of processes) can fail arbitrarily long (drop unbounded

# of messages)

7.5.2 Goal 1: Termination, Agreement, Validity

- Termination: all nodes eventually decide
- Agreement: all nodes settle on one same value
- Validity: if all started as same value, must settle on that value.

1. There is no deterministic algorithm

    1. Suppose there is one. Two processes with a eternally failing channel. Assume both have input 1, then has to conclude with 1.

process	input	conclusion
A	1	1
B	1	1

    2. Suppose process B has input 0. To A, this is indistinguishable from B having input 1 since they cannot communicate. Thus B will still conclude 1.

process	input	conclusion
A	1	1 (indistinguishable)
B	0	1

    3. By Agreement, B will conclude as 1 too, force by A’s ignorant conclusion.

process	input	conclusion
A	1	1
B	0	1 (Agreement)

    4. Now suppose A has input 0 as well. To B, this is indistinguishable from A having 1 (remember by 2, 3, A having 1 forces B to conclude 1) so it will still conclude with 1.

process	input	conclusion
A	0	1
B	0	1 (indistinguishable)

    5. Then A will be forced to conclude with 1, violating Validity. Contradiction.

process	input	conclusion
A	0	1 (validity)
B	0	1

7.5.3 Goal 2: T, A, Weakened Validity

- Since validity was violated just now, we try to weaken it. Is consensus possible now?
- If all start from 0, should settle on 0
- If all start from 1, then must settle on 1 only if no message is lost

1. Still no deterministic algorithm.
   - Lemma: if two processes start with 1, and one process's last message is lost, none detects it. Then the one who lost, B, didn't know it lost, hence is **indistinguishable** from nothing is lost and must settle on 1. By agreement, A should settle on 1 as well.
   - Using the lemma above, we can still chain indistinguishability from 0, 0 to 1, 1 and force a contradiction (settle on 0, 0 given 1, 1, when all messages are lost)
   1. Proof of no deterministic algorithm

      Essentially the same as in the previous Goal, but start with 0,0. Achieve contradiction not by validity (we weakened it thus would not be contradicted), but consider an undetected loss of last message, which has to conclude with 1,1.

7.5.4 Goal 3: T, Limited Agreement, Weakened Validity

- Limited Agreement: all nodes decide on the same value with probability = (1-ϵ)

1. The concept of adversary

   We imagine the message losses are not just random, but as bad as possible, as if designed by an adversary to destroy our plans for consensus. Then we do a worst case analysis against this adversary.

2. Randomized Algorithm that works with error (1/r)
   1. Setup: We have processes P1, P2, etc.
      - All random choices not known by adversary beforehand.
      - Let the total number of rounds to be `r`.
      - Each process has an **input**, 0 or 1. Each process has a value called **level**, initialized to 0.
      - P1 sets the a **bar** randomly between 1 and r, inclusive. Every process tries to increment **level** so that they reach the **bar** over the `r` rounds.
   2. Broadcast messages and receive messages for r rounds. Each message contain **bar** (if you know it), **input**, and own **level**. Upon receive, update own **level** to `l+1` if everyone is at least level `l`.
   3. We can inductively show that every pair of levels must differ by at most one at every round.
   4. Once r rounds are over, decide on 1 IFF you know that everyone has input 1 AND you know bar (trivial for P1) AND level >= bar.
   1. Analysis
      1. Termination (obvious by `r` rounds)
      2. Agreement with probability (1-1/r)
         - Since not every process's **level** will reach **bar**, not everyone can decide

      correctly (eg, should decide 1 but limited by insufficient level). We then analyze what is the probability for these events.

         - Error only happens if P1 decide on 1 while P2 decide on 0 (WLOG).
         - Denote P1's level as L1, P2's level as L2.
         1. Scenario: One process decides 1, the other process decides 0
            1. Case 1: P1 hears P2, but P2 never heard from P1 at all

               - Then P2 must have level 0 (L2 = 0). It never heard from P1 and does not know bar, hence will conclude 0.

process	level	decision
P1		
P2	0	0

- P1 hears P2 so must have level 1 (L1 = 1). To be an error case, it must decide 1.

process	level	decision
P1	1	1
P2	0	0

- What do we know, now P1 decides 1? All conditions below must be fulfilled:
  1. Knows bar (no new info, P1 set the bar)
  2. All inputs are 1
  3. 1 >= bar. Since bar is at least 1, bar = 1.

1. Conclusion: only happens when bar = 1, probability = 1/r

2. Case 2: P2 hears P1, but P1 never heard from P2 at all

- Then P1 must have level 0 (L1 = 0). It never heard from P2 and does not know P2's input, hence will conclude 0.

process	level	decision
P1	0	0
P2		

- P2 hears P1 so must have level 1 (L2 = 1). To be an error case, it must decide 1.

process	level	decision
P1	0	0
P2	1	1

- What do we know, now P2 decides 1? All conditions below must be fulfilled:
  1. Knows bar (no new info, P2 heard from P1)
  2. All inputs are 1
  3. 1 = L2 >= bar. Since bar is at least 1, bar = 1.

1. Conclusion: only happens when bar = 1, probability = 1/r

3. Case 3: P1 did not hear from P2 and vice versa

   This case would not cause error, since they will both conclude 0.

4. Case 4: P1 heard from P2 and vice versa
   - The only violatable condition before concluding 1, is that `level<bar`.
   - Their **level** difference is at most one (by lemma).
   - Probability is `1/r`, when the **bar** is set to the maximum of the two.

2. Conclusion: error probability is 1/r.

3. Weakened Validity

        - suppose everyone starts with 0, no one can conclude 1 therefore everyone will conclude 0.
        - suppose everyone starts with 1.
            - suppose no message lost, everyone will reach **bar** in `r` ($r \geq bar$) rounds. In fact, everyone will have **level** = `r`. Everyone will conclude 1.
            - suppose message lost, then can decide on anything.
3. Error probability of 1/#rounds is a lower bound.
    - proof can be found in Lynch.

7.6 Version 3: (Asynchronous) Node crash failures

7.6.1 Setup

- Nodes can crash (indefinitely), but channels are reliable
- Asynchronous: message delay is unbounded
- Impact: can no longer define a round

7.6.2 Fischer-Lynch-Paterson (FLP) Impossibility Theorem

Statement: the distributed consensus problem under the asynchronous timing model is impossible to solve, even with a **single** node crash failure.

- Fundamental reason: the protocol is unable to accurately detect node failure.

7.7 Version 4: (Synchronous) Byzantine Failures

7.7.1 Failure Model

Each node can "lie" about its input.

7.7.2 Modified goals - everything only applies to non-faulty nodes

7.7.3 A simple (unsuccessful) attempt

1. (Byzantine fault) A tells B 1, A tells C 0. B tells everyone 1. C tells everyone 0.

node	received
A	1, 1, 0
B	1, 1, 0
C	0, 1, 0

2. Anyone could be lying. C decides B is lying -> it decides on 0. Similarly, B will decide on 1, violating Agreement.
3. Furthermore, if you cross-check, you are not sure if the person you cross-check with is reliable.

7.7.4 Theorem: Byzantine Consensus Threshold

- Theorem: if number of processes `n`, number of byzantine failures `f`, if `n<=3f`, then any protocol cannot reach consensus.
- Proof: omitted.

7.7.5 A protocol for # nodes n >= 4f+1

1. Definitions
   - We have *phases*.
   - Each phase has a *coordinator* which sends a proposal to all processes.
   - If everyone decides, we have an agreement.
   - A phase is a *deciding phase* if the coordinator is nonfaulty.
   - Each phase has a 3 rounds: broadcast, coordinator, decision.
2. Protocol
   1. Setup: maintain everyone's value initialized to 0, my value = my input.
   2. Repeat the following steps (called a phase) for `f+1` times:
      1. Round 1: all-to-all broadcast. Send my input to everyone (including myself). Record all msg received (if msg is non-binary or null, set it to 0). If there is a majority (occurrence > n/2) value, let `proposal` be that value, otherwise `proposal` is 0.
      2. Round 2: coordinator. if I am the coordinator (the ith phase has the ith node as coord), send proposal to all, otherwise receive a proposal.
      3. Round 3: if there is an overwhelming majority in V (already determined in R1), ie > n/2 + f, set my value to that majority value, otherwise set it to the proposal.
   3. After phases end, conclude with my value.
3. Correctness - invariance for every round
   1. Lemma 1: all non-faulty processes retain `y` if all their values are `y`

      Since $n \geq 4f + 1$, we must have $n - f \geq 3f + 1 \geq (4f + 1)/2 + f = n/2 + f$. This means the non-faulty nodes must be a overwhelming majority. Suppose they all have the same value at the start of phase `k`, they must retain these values at end of phase `k` forced by round 3.

   2. Lemma 2: if coord is nonfaulty, then all nonfaulty processes have the same value after a round k.

      The coordinator can propose two values:

      1. Proposes some `x`. `x` occurs MORE THAN `n/2` times on coordinator process. Worst case, `f` out of `n/2` are faulty. These MORE THAN `n/2-f` non-faulty nodes must broadcast exactly these values to others, thus on any other nodes, they cannot have another value `y` having the overwhelming majority (`y` can only have LESS THAN `n/2+f` votes) in R3 and can thus only accept the proposal.
      2. Proposes 0. This means out of all the values coord sees, none exceed `n/2`. It means none of these values will reach overwhelming majority MORE THAN `n/2+f` since the amount can differ by atmost f.
   3. Termination - we let there be f+1 phases. No wait => termination.
   4. Validity - by lemma 1, the invariance make them decide its input.
   5. Agreement
      - `f+1` phases means at least one good phase.
      - The one good phase triggers lemma 2.
      - Lemma 2 triggers lemma 1, agreeing on the value as per lemma 1.

Author: Tan Yee Jian

Created: 2021-05-04 Tue 08:13