CS 194: Elections, Exclusion and Transactions

Scott Shenker and Ion Stoica
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1776

1

Finishing Last Lecture

- We discussed time synchronization, Lamport clocks, and vector clocks
  - Time synchronization makes the clocks agree better
  - Lamport clocks establish clocks that are causally consistent
    - But they leave too much ambiguity
  - Vector clocks tighten up ambiguity by weaving much finer web of causality
    - Lots of overhead
- I'll now finish up the material on global state

2

Global State

- Global state is local state of each process, including any sent messages
  - Think of this as the sequence of events in each process
  - Useful for debugging distributed system, etc.
- If we had perfect synchronization, it would be easy to get global state at some time t
  - But don't have synchronization, so need to take snapshot with different times in different processes
- A consistent state is one in which no received messages haven't been sent
  - No causal relationships violated

3

Method #1: Use Lamport Clocks

- Pick some time t
- Collect state of all processes when their local Lamport clock is t (or the largest time less than t)
- Can causality be violated?
- A violation would required that the receipt of the message is before t and the sending of it is after t.

4

Method #2: Distributed Snapshot

- Initiating process records local state and sends out "marker" along its channels
  - Note: all communication goes through channels!
  - Each process has some set of channels to various other processes
- Whenever a process receives a marker:
  - First marker: records state, then sends out marker
  - Otherwise: records all messages received after it recorded its own local state
- A process is done when it has received a marker along each channel; it then sends state to initiator
  - Can't receive any more messages

5

Why Does This Work?

- Assume A sends message to B, but in the snapshot B records the receipt but A does not record the send
- A's events: receive marker, send message out all channels, then send message to B
- B's events: receive message from A, then receive marker
- This can't happen! Why?

6

What Does This Rely On?

- Ordered message delivery
- Limited communication patterns (channels)
- In the Internet, this algorithm would require n^2 messages

7

Lamport Clocks vs Snapshot

- What are the tradeoffs?
- Lamport: overhead on every message, but only on the messages sent
- Snapshot: no per-message overhead, but snapshot requires messages along each channel
  - If channels are limited, snapshot might be better
  - If channels are unlimited, Lamport is probably better

8

Termination Detection

- Assume processes are in either a passive state or an active state:
  - Active: still performing computation, might send messages
  - Passive: done with computation, won't become active unless it receives a message
- Want to know if computation has terminated
  - all processes passive
- Not really a snapshot algorithm

9

Termination Detection (2)

- Send markers as before (no state recording)
- Set up predecessor/successor relationships
  - Your first marker came from your predecessor
  - You are your successor's predecessor
- Send "done" to predecessor if:
  - All your successors have sent you a "done"
  - You are passive
- Otherwise, send "continue"
- If initiator gets any "continue" messages, resends marker
- If initiator gets all "done" messages, termination

10

Comments

- Few of these algorithms work at scale, with unreliable messages and flaky nodes
- What do we do in those cases?

11

Back to Lecture 7

- Elections
- Exclusion
- Transactions

12

Elections

- Need to select a node as the "coordinator"
  - It doesn't matter which node
- At the end of the election, all nodes agree on who the coordinator is

13

Assumptions

- All nodes have a unique ID number
- All nodes know the ID numbers of all other nodes
  - What world are these people living in???
- But they don't know which nodes are down
- Someone will always notice when the coordinator is down

14

Bully Algorithm

- When a node notices the coordinator is down, it initiates an election
- Election:
  - Send a message to all nodes with higher IDs
  - If no one responds, you win!
  - If someone else responds, they take over and hold their own election
  - Winner sends out a message to all announcing their election

15

Gossip-Based Method

- Does not require everyone know everyone else
- Assume each node knows a few other nodes, and that the "knows-about" graph is connected
- Coordinator periodically sends out message with sequence number and its ID, which is then "flooded" to all nodes
- If a node notices that its ID is larger than the current coordinator, it starts sending out such messages
- If the sequence number hasn't changed recently, someone starts announcing

16

Which is Better?

- In small systems, Bully might be easier
- In large and dynamic systems, Gossip dominates
- Why?

17

Exclusion

- Ensuring that a critical resource is accessed by no more than one process at the same time
- Centralized: send all requests to a coordinator (who was picked using the election algorithm)
  - 3 message exchange to access
  - Problem: coordinator failures
- Distributed: treat everyone as a coordinator
  - 2(n-1) message exchange to access
  - Problem: any node crash

18

Majority Algorithm

- Require that a node get permission from over half of the nodes before accessing resource
  - Nodes don't give permission to more than one node at a time
- Why is this better?
- N=1000, p=.99
  - Unanimous: Prob of success = $4x10^{-5}$
  - Majority: Prob of failure = 10^{-7}
  - 12 orders of magnitude better!!

19

Interlocking Permission Sets

- Every node I can access the resource if it gets permission from a set V(I)
  - Want sets to be as small as possible, but evenly distributed
- What are the requirements on the sets V?
- For every I,J, V(I) and V(J) must share at least one member
- If we assume all sets V are the same size, and that each node is a member in the same number of sets, how big are they?

20

Transactions

- Atomic: changes are all or nothing
- Consistent: Does not violate system invariants
- Isolated: Concurrent transactions do not interfere with each other (serializable)
- Durable: Changes are permanent

21

Implementation Methods

- Private workspace
- Writeahead log

22

Concurrency Control

- Want to allow several transactions to be in progress
- But the result must be the same as some sequential order of transactions
- Transactions are a series of operations on data items:
  - Write(A), Read(B), Write(B), etc.
  - We will represent them as O(A)
  - In general, A should be a set, but ignore for convenience
- Question: how to schedule these operations coming from different transactions?

23

Example

- T1: O1(A), O1(A,B), O1(B)
- T2: O2(A), O2(B)
- Possible schedules:
  - O1(A),O1(A,B),O1(B),O2(A),O2(B) = T1, T2
  - O1(A),O2(A),O1(A,B),O2(B),O1(B) = ??
  - O1(A),O1(A,B),O2(A),O1(B),O2(B) = T1, T2
- How do you know? What are general rules?

24

Grab and Hold

- At start of transaction, lock all data items you'll use
- Release only at end
- Obviously serializable: done in order of lock grabbing

Grab and Unlock When Not Needed

- Lock all data items you'll need
- When you no longer have left any operations involving a data item, release the lock for that data item
- Why is this serializable?

Lock When First Needed

- Lock data items only when you first need them
- When done with computation, release all locks
- Why does this work?
- What is the serial order?

Potential Problem

- Deadlocks!
- If two transactions get started, but each need the other's data item, then they are doomed to deadlock
- T1=O1(A),O1(A,B)
- T2=O2(B),O2(A,B)
- O1(A),O2(B) is a legal starting schedule, but they deadlock, both waiting for the lock of the other item

Deadlocks

- Releasing early does not cause deadlocks
- Locking late can cause deadlocks

Lock When Needed, Unlock When Not Needed

- Grab when first needed
- Unlock when no longer needed
- Does this work?

Example

- T1 = O1(A),O1(B)
- T2 = O2(A),O2(B)
- O1(A),O2(A),O1(B),O2(B) = T1,T2
- O1(A),O2(A),O2(B),O1(B) = ??

31

Two Phase Locking

- Lock data items only when you first need them
- After you’ve gotten all the locks you need, unlock data items when you no longer need them
- Growing phase followed by shrinking phase
- Why does this work?
- What is the serial order?

32

Alternative to Locking

- Use timestamps!
- Transaction has timestamp, and every operation carries that timestamp
- Serializable order is timestamped order
- Data items have:
  - Read timestamp tR: timestamp of transaction that last read it
  - Write timestamp tW: timestamp of transaction that last wrote it

33

Pessimistic Timestamp Ordering

- If ts < tW(A) when transaction tries to read A, then abort
- If ts < tR(A) when transaction tries to write A, then abort
- But can allow
  - ts > tW(A) for reading
  - ts > tR(A) for writing
- No need to look at tR for reading or tW for writing

34

Optimistic Timestamp Ordering

- Do whatever you want (in your private workspace), but keep track of timestamps
- Before committing results, check to see if any of the data has changed since when you started
- Useful if few conflicts

35