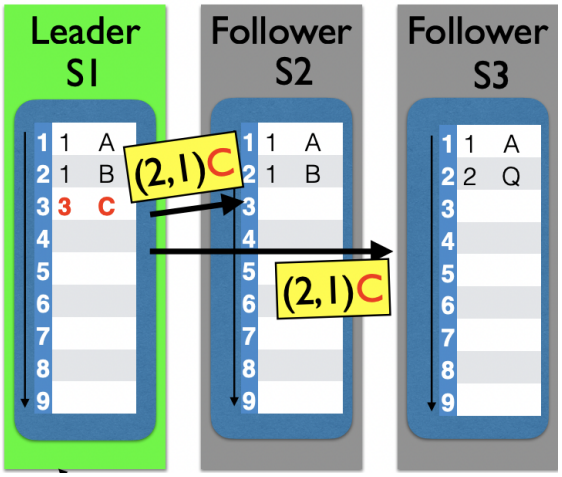
CAS CS 350
Lec 12

Raft

1. Overview
    a. Raft → algorithm that is distributed consensus
    b. Use to implement replicated state machine (replication approach to make machines fault tolerant)
        i. Use either state transfer or replicated machine to do primary backup for fault tolerance
        ii. Assigning the same task to multiple workers (mapReduce) and using first result that is available → another fault tolerance method
    c. Big picture of raft
        i. Bunch of servers, bunch of clients
        ii. Servers correspond to system that we want to make it fault tolerant (x86 command)
        iii. The server stores log → each log of bunch of entries
            1. Entries store commands from clients
            2. Logs store commands by the server to make it fault tolerant
                a. ex) key-value pairs → get command
            3. Need to store the read commands in the log as well because we want to support strong consistency
    d. In raft, client cannot see outdated version of the log
    e. Linearlize ability → strongest consistency model you can have (client views the most updated data and never see outdated data)
    f. We can store whatever command we want in the log
    g. Terms, index → metadata
        i. Sequence numbers used by Raft itself
    h. In raft, all client replications must go through the leader → cannot see outdated data (if there is no leader due to election failure, the system is unavailable and the client request will return an error and asks the client to retry)
    i. Goal is that servers agree on the identical log (servers need to decide same version of the log by only sending messages to each other)
2. What are we hoping for?
    a. Tolerate failure of a minority of followers
    b. Convergence to one version of the log
    c. Only execute client requests (and thus reply to client) when it is “committed” - can’t go back and unexecuted or un-tell client a response

d. When you want to avoid duplicates within the log of raft, you can build other logic on top of raft to prevent it → if the client makes duplicate request, send the previous version (raft does not prevent duplicates by itself, the only problem it solves is fault tolerance of minority and converge up to one identical log)

3. No Leader failure
   a. 

   b. C is the command that we want to append
   c. 2 is the index number of the previous command
   d. 1 is the index number of the previous term
4. 2 common divergence
   a. When new leader is selected, it forces followers to convert to leader's version of the log
   b. When new leader is elected, it must never be left behind (leader must always have all committed entries in its log)
      i. If candidate does not have all committed entries, it cannot be the leader
5. What happens if an election fails? (Everyone detects failure and stars an election voting for themselves → can't vote for anyone else then and no-one can get majority)
   a. Split brain problem → two parts that are disconnected (each part believes that the other part is down)
6. Failed elections
   a. Nothing is particularly wrong a new term will be initiated and an associated election (system just becomes unavailable and will be timed out until there is a leader)
   b. So how do we ensure there is more than zero leader ever elected
   c. Probabilistic approach: can't stop any particular election from failing but with high probability some election will succeed
   d. Random time out: if the server does not receive majority during the random time out (each server has its own random time out)
      i. It increases the term
      ii. Resets the time out election

        iii. Servers vote again
        iv. It votes for itself (some servers will have a higher probability to be elected depending on the time out)
7. Randomized delays
    a. Split votes will unlikely happen repeatedly
        i. When a server wants to become a candidate it picks a random number and wait
        ii. If no-one was elected in that amount of time then actually start the election for oneself and ask for votes
    b. There is no upper bound in the number of terms
8. Intuition for random delays
    a. Leader crashes here and follower's heartbeat time expires
    b. All followers start the new election (each follower sets a random timeout → even if everyone votes for itself in the first election, in second election, it has more possibility to become the leader since it has the shortest timeout
    c. When a server initiates a new election, it votes for itself and requests vote from other servers (number of term increases)
    d. In this case, follower0 and follower1 already voted → they cannot vote again due to the majority rule
        i. If a server receives request vote from other server even if they voted, if they visualize that the term number is increased, it goes to a follower state automatically → follower2 will become the leader
9. Notice another pattern in the design
    a. Breaks hard problem down and applies two different approaches that complement each other
        i. Critical safety property (correctness) - at most one leader - addressed by a hard mechanism (must have a majority) that can fail leading to retry, leaders should not be left behind
        ii. Performance/Liveness - softer probabilistic scheme - retry and hopefully eventually succeed - no impact on correctness (based on probability)
    b. A way of thinking: separate safety from performance/liveness