Lecture 3

SMR with Crash Faults in Synchrony

- recall from Lecture 2 definition of the SMR problem, definitions of consistency and liveness
- recall from Lecture 2 the definition of crash faults (easy mode(), Byzantine faults (hard mode), synchronous network (easy mode), asynchronous network (hard mode), and the road map from easy/easy to hard/hard
- goal for this lecture:
  - get a feel for the challenge of crash faults
  - solve SMR (i.e., consistent and live protocol) in easy/easy mode (crash faults, known bound Δ on message delays)
  - discuss challenges of and compromises around asynchrony
- starting point: Protocol A from last time:
  - "view" = Δ consecutive timeslots
  - validators take turns as leader (one per view)
  - in view $v = 0, 1, 2, \ldots$:
    - at time $\Delta \cdot v$, leader assembles block B (all not-yet-included txs that it knows about)
    - at time $\Delta \cdot v$, leader sends B to all other validators
    - at time $\Delta \cdot v + \Delta$, all validators append B (if any received form leader) to their local chains/histories
- question: what could go wrong? (might seem like it would be irrelevant if non-leaders crash, and if the leader crashes everybody just skips that view, no harm no foul)
- problem: leader might crash after sending B to some but not all other validators (would violate consistency, why?)

- example execution in which, due to issue above, consistency is violated (see slides)
- fix:
  - validators will update the next leader as to their current history
    - to make sure leader is as up-to-date as everyone else
    - doubles the length of the view, as now have non-leader-to-leader communication (catch-up messages) followed by the opposite (leader's proposal)
  - validators pass around entire history/chain, not just latest block
    - will see how to make this practical using cryptographic commitments in Part II
    - point: due to possibility of a sequence of leader crashes, a validator might learn about a number of new blocks at the same time
- blending these ideas into Protocol A brings us to Protocol B:
  - "view" $= 2\Delta$ consecutive timeslots (extra Δ so that there's time for non-leaders to update leader before leader makes a proposal)
  - validators take turns as leader (one per view)
  - each validator maintains a local chain C_i
  - in view $v = 0, 1, 2, \ldots$:
    - at time $2\Delta \cdot v$, each validator i sends the leader ℓ of the view its current chain C_i
    - at time $2\Delta \cdot v + \Delta$:
      - let C denote longest C_i received by ℓ in this view
      - leader assembles block B (all not-yet-included (in C) txs that it knows about)
      - leader sends $C^* := (C, B)$ to all validators
    - at time $2\Delta \cdot v + 2\Delta$:
      - if validator i receives a new chain C^* from the leader, it updates $C_i := C^*$
- show picture of communication pattern of one view of Protocol B
- show picture of communication pattern of multiple views of Protocol B
- give example execution (already a surprising number of different ways that things might play out)
  - e.g., at end of view with a faulty leader, non-faulty validators may have different local chains (no consistency violation yet, but entering dangerous territory)

  - e.g., might have one version B_3 of block number 3 that gets forgotten (because all validators that knew about it have crashed), and subsequently a new version B_3' is proposed by a non-yet-crashed validator (again, seems to be drifting dangerously close to a consistency violation)
  - in example, never actually wind up with a consistency violation — could this be true in general?
    * should demand a proof—in general, a distributed protocol without a correctness proof is probably buggy
    * and bugs in a successful and Internet-scale consensus protocol are likely to surface (runs for multiple years, widely varying Internet conditions and workloads)
    * proofs play an unusually central role in distributed computing—generally have to design a protocol and its correctness proof at the same time, each deeply informing the other
    * only by seeing the correctness proofs can you understand why e.g. Paxos/Raft and Tendermint work the way that they do

- proof of consistency:
  - recall: consistency is equivalent to all (not-yet-crashed) validators' local chains being prefixes of a common chain (otherwise the superposition of all the local chains has a fork, representing a consistency violation)
  - claim: at each timeslot, the chains of the not-yet-crashed validators are consistent

    [exercse: note chains of the crashed validators need not be consistent with the uncrashed ones, why?]
  - proceed by induction on timesteps (certainly true at the beginning)
  - only time something could go wrong is at the end of a view, when validators might update their local chains (e.g., for view v, at timestep $2\Delta \cdot v + 2\Delta$)
  - for each view v, by the inductive hypothesis, all C_i's sent to the leader are consistent with each other (i.e., all prefixes of a common chain)
    * these were the local chains of all not-yet-crashed validators at timestep $2\Delta \cdot v$ (at which the inductive hypothesis applies)
  - thus C, the longest of these C_i's, will extend all the C_i's (in fact, C is exactly the common chain that all the C_i's are prefixes of)
  - since C^* is just C with an extra block (B) tacked on the end, it will extend all of the C_i's
  - thus, no matter which validators update their local chains to C^* (i.e., no matter which validators receive the leader's proposal before the leader possibly crashes), the inductive statement will continue to hold

- proof of liveness:
  - suppose tx z known to some non-faulty validator i at timestep t
  - let v be the next view where i is the leader (exists, why?)
  - i's proposal in that view will include z (if nothing else, in the new block B)
  - since i non-faulty, by end of view v, all uncrashed validators will have adopted i's proposal and included z in their local chains
- key takeaways/design patterns from Protocol B and its analysis (will recur in the Paxos/Tendermint, and also many modern blockchain protocols):
  1. views = repeated attempts to finalize new txs
  2. leaders = coordinate the txs proposed in each view
     - chosen e.g. round-robin, or randomly
  3. view may end with non-faulty validators in different states
     - possibly due to a faulty leader and/or due to asynchrony
     - leader of next view may need to "clean up the mess" left by the previous view
  4. leader should be as up-to-date as all non-faulty validators
     - otherwise, an out-of-date leader might (despite the best of intentions) propose a chain that conflicts with those of up-to-date validators
     - worse, this proposal might be adopted by other out-of-date validators, leading to a consistency violation with the previously up-to-date validators
  5. distributed computing is hard! (even the playground of crash faults and bounded message delays is non-trivial)
     - you may experience this yourself on the homeworks!
- next challenge: asynchrony
  - synchrony assumption is pretty strong, would prefer protocols robust to unexpected network outages/attacks
- question: is Protocol B still consistent and live if some messages might get delayed more than Δ timesteps?
- answer: no. reason: leader may not hear about all non-faulty C_i's by the time it makes a proposal
  - have the issues with an out-of-date leader mentioned above. e.g., if some non-faulty validator has local chain $B_1 \rightarrow B_2 \rightarrow B_3$, the leader only knows about $B_1 \rightarrow B_2$ (because catch-up message from the former validator is delayed), leader may propose $B_1 \rightarrow B_2 \rightarrow B_3'$, which might be adopted by some other validator that knows only about $B_1 \rightarrow B_2$ (which is a consistency violation)

- will need a more sophisticated protocol to deal with asynchrony (see Lecture 4 for the solution, which involves adding additional friction to validators finalizing new txs and to leaders making proposals, and also a new assumption that a strict majority of the validators are non-faulty)
- question: how should we model asynchrony?
- bad answer: make Δ really big (basically avoid the problem, wind up with stupid/impractical protocols that sit around idly most of the time)
- ambitious answer: no assumptions on message delays at all (other than eventually delivery, which is presumably necessary for anything to be possible)
  - called the *asynchronous model*
- problem: (FLP theorem) even with the threat of a single crash fault, can't solve SMR with an asynchronous network (exact statement is subtle and proof is quite non-trivial, see Bonus Lecture 1 for a detailed discussion); in the main lectures, we will take this result on faith
- perspective: the point of an impossibility result is not to give up, it's to provide guidance as to the sort of compromises you'll need to make to make progress (i.e., places limits on the best-case scenario)
- possible compromises:
  1. pull back from asynchrony to "partial synchrony" (next lecture, sweet spot between synchrony and asynchrony)
     - partial synchrony assumption is weak enough to be practically relevant, but strong enough that can design protocols with satisfying guarantees
  2. solve a problem easier than SMR (e.g., with relaxed consistency requirement)
     - this is interesting and relevant to blockchains (e.g., for unrelated payments like Alice $\rightarrow$ Bob and Carol $\rightarrow$ David, doesn't matter is different validators disagree on their orcer), but we won't have time for it, alas
     - would make for a good project
  3. use randomized protocols to solve SMR with high probability (FLP does not rule this out and, indeed, such protocols exist)
     - big literature here, but mostly academic, limited practical impact (at least so far)
     - again, would make for a good project