What is practical BFT?

- Byzantine consensus protocol
  - Byzantine in the sense of BGP
  - Consensus in the sense of paxos
    - Sequence of operations
    - “agents” [replicated state machines] must agree upon operations and their order
    - clients are more or less trusted, and can suggest any operation
      - with the caveat that agents must agree, so a malicious client can still do damage but it is consistent damage
  - mechanisms in place to prevent faulty primary from preventing forward progress
    - Byzantine failures can be designed to stall
    - In paxos, liveness is only threatened by network delays and timing coincidences from multileaders
    - In PBFT, have to prevent “malicious” timing attacks

- Scheme:
  - Client sends request to primary, hears back directly from backups
  - If doesn’t hear back soon enough, then client broadcasts directly to backups, which relay to primary
  - Because ordering is important, need to agree on order
    - Hence, have primary set order
    - Requires PBFT to also maintain consensus on who is primary
    - Basically a fault-tolerant token-holder subconsensus problem

- Assumes:
  - Valid signatures on all replicas
  - A non-faulty replica cannot have its signature forged by somebody else
  - Some bounds on response times to ensure liveness
    - Still possible to not hit consensus, but really only in case that responses are delayed arbitrarily, i.e., no recovery happens
  - At most k faults for 3k+1 replicas
    - K+1 assertion of same value proves at least one non-faulty replica asserts that value
    - 2K+1 votes for same value convinces all replicas that this value has majority within non-faulty nodes and should be considered true
Assume everybody agrees on a view:

- Normal case operation:
  - Client sends signed \(<\text{REQUEST}, \text{op}, T, \text{client}>\) message to primary
    - \(T = \text{timestamp}\)
    - Primary interacts with backups
  - Backups eventually send response messages to client
    - \(<\text{REPLY}, v, T, c, i, r>\text{signed}_I\)
      - \(v = \text{“current view”}\)
      - \(I = \text{backup number}\)
      - \(R = \text{response}\)
  - If client sees \(f+1\) replies with same \(T, R\), and with valid signatures, it accepts the result
  - If client times out before seeing replies, it retransmits REQUEST by broadcasting directly to all replicas
    - This is what helps kickstart a viewchange later
  - If client still times out, gives up! No consensus possible, and not clear if operation succeeded.

- OK, so drill down into the “Primary interacts with backups”
  - Three phase operation: pre-prepare, prepare, and commit
    - Primary multicasts request to backups, preserving signature
      - \(<<\text{PRE-PREPARE}>, v, n, d>\text{signed}_p, m>\)
        - \(n = \text{a sequence number}\)
        - \(d = \text{digest of message [why?]}\)
          - so that message could be sent using different protocol
          - so that primary doesn’t have to sign entire message
    - Backup accepts preprepare message iff:
      - signatures in request and preprepare are correct
      - \(d = \text{digest for } m\)
      - backup is actually in view \(v\) [why?]
        - so ordering is set by a single primary
      - it hasn’t accepted prepare for view \(v\) and sequence \(n\) before
        - sequence number between low, high water mark
          - so primary can’t exhaust sequence number space
    - Outcome of preprepare is that backups know they need to kibitz with each other to see if enough of them have agreement
    - Prepare: get replicas to make an order stable
      - each backup multicasts \(<\text{PREPARE}, v, n, d, i>\text{signed}_I\) to all other replicas, and adds both preprepare and its sent prepare messages to log
      - each backup accepts PREPARE messages and adds those to log too, if:
• signatures are correct, view number matches local view, and sequence number between watermarks
• thus, if anybody disagrees on view, everybody will discover this
  ▪ predicate prepared \( (m,v,n,i) \) true iff replica \( I \) has inserted into its log (request \( m \), preprepare for \( m \) with view \( v \) and seq # \( n \), and \( 2f \) prepares from different backups that match the preprepare)
  ▪ thus, if prepared \( (m,v,n,I) \) is true, all replicas will eventually agree upon order of messages, and validity of messages
    o because all non-faulty replicas will eventually have the prepared predicate as true

o commit: make order stable across views
  ▪ a replica (including primary) multicasts a commit message
    • \(<\text{COMM}, v, n, D(m), I>\)signed_I
  ▪ when prepared \( (m,v,n,I) \) becomes true
  ▪ replicas accept commit messages and insert in log provided everything matches up
  ▪ two new predicates:
    • committed \( (m,v,n) \): true iff prepared \( (m,v,n,I) \) is true for a set of \( f+1 \) non-faulty replicas
      o which is what you want to guarantee that those non-faulty replicas will send response to client
    • committed-local \( (m,v,n,I) \) is true iff prepared \( (m,v,n,I) \) is true and \( I \) has accepted \( 2f+1 \) commits (including maybe its own)
      o if committed-local is true for some \( I \), then committed is true
      o if committed-local is true for some \( I \), then it will become true for at least \( f+1 \) non-faulty replicas
  ▪ a replica executes operation requested by \( m \) once committed-local is true and all previous op sequence numbers have been executed
• messages can commit out of order, that’s ok

The wrinkles:
• garbage collection- when can you eliminate stuff from logs?
  o A replica can eliminated a message’s gunk from log when that replica is convinced that at least \( f+1 \) non-faulty replicas have executed the operation
• Intuition: periodically generate checkpoints of service state
  o Prove that checkpoint is correct
    ▪ If can prove it, can eliminate messages behind checkpoint
    ▪ Also, can use that checkpoint to recover another replica
  o Proof:
    ▪ Snapshot-like algo
    ▪ At some event trigger (like sequence number = 0 mod 100) all replicas issues \(<\text{CHECKPOINT},n,d,I>\)signed_I message and sends to everybody
• N is latest sequence number in checkpoint state
• D = digest of checkpoint state
• Checkpoint must be put somewhere on stable storage
  ▪ Everybody collects these checkpoint messages in their logs
  ▪ When somebody has 2f+1 of them, that person has proof.
  o Checkpoints becoming stable (proven) are also used to advance high/low water marks
• can also use checkpoints for view changes
  o basically, when anybody wants to advance view (because it believes primary has conked out), that replica sends out a view-change message that contains new view number, and a proof of last stable checkpoint it knows about
  o also includes “leftover” state of prepared messages that aren’t in the checkpoint
    ▪ since those are used to commit these “leftovers”
  o need a bunch of people (2f+1) to independently decide to send out view-change messages before a view change happens
    ▪ this prevents starvation through frequent view change
  o when view change is initiated, the initiator stops processing non view change messages (viewchange, checkpoint, and new-view)
  o primary for new view terminates the viewchange protocol

Benchmarks

• lies, damn lies!
• Ran Andrew benchmark with a single client
• This means that:
  o Each operation does full RTT before next is issued
  o Means that server is underutilized
  o Means that overhead of crypto isn’t included
  o Only seeing effect of extra round-trip of protocol