PBFT

Castro and Liskov. Practical Byzantine Fault Tolerance
(Not an easy read)

Reminders

- Post your project ideas to piazza/slack.
- **Proposal drafts due next Friday (Oc 9th)**

PBFT

- What’s a byzantine fault?
    - Includes *software bugs*: e.g., critical data might be manipulated incorrectly.
    - Hardware faults
    - Malicious attacks against f nodes in the system
- Why should we care about such faults?
    - Generally rare if you “trust” your machine, but if it is networked, then someone might be “borrowing” your machine
    - Networking changes the equation: bugs will leak out into the rest of the system, bugs can enter from outside (including malicious actors)
    - Today BFT is a huge topic (also blockchains)
    - Distributed systems more complex (concurrency) => more likely to be buggy => more likely a concern to protect yourself from
- Why do we *not* care about such faults?
    - Company would find these expensive to handle! Let the clients suffer the consequences

PBFT

- What is PBFT attempting to do?
- Builds a protocol that is practical to implement, provides BFT guarantees.
- Practical?
    - Recovers from leader/replica faults
    - Reasonable performance (proof of concept for NSF with 3% overhead)
    - NSF is the ultimate practicality test (at the time): killer distributed system application. If it works on NSF .. assumption is it works on anything.
        - Nicely aligns with low state requirements on server (replicas): snapshot is ‘small’, state is on DISK for NSF.
        - NSF is mostly deterministic (except for pesky timestamps on files)
        - **Any application needs a file system. By handling NSF, they can be used by any app. NSF is well known**
        - NSF workflow matches RSM nicely: request from client to server (which is replicated), which then executes, and replies back to client
    - Contrasts with existing work (at the time) that was more theoretical
    - Assumes a very weak network model: fully async, like Paxos (always safe, usually live), reordering and duplication ok

PBFT

- *The interesting technical bits you enjoyed*
- Builds on/inspired by **Paxos** (more phases), leader based model (more efficient than Paxos; more like view-stamped replication), null-based filling of slots (null proposals).
- *BFT protocols are sort of like a special case of distributed consensus*
- Low and high watermarks for sequence numbers (with strict invariants; related to snapshots)
- Guarantees total ordering even with view changes
- View change protocol that is built-in: transition from view v to view v+1 requires overlap of nodes that share state
- Practical: Retain the current view as long as possible (timer that is maintained by each replica, with 2x doubling of the timer to accommodate network async)
- Practical: Partition the distribution of data from the voting (meta-data consensus): use different transports. And they connect data + meta-data with message digests.
- Practical: Checkpoints allow replicas to clear the log of messages (garbage collection). They also determine the low watermark value. Agreement on checkpoints. Async protocol (runs in the background concurrently with mainline protocol). Also used to update new/old nodes to bring them up to date.

PBFT

- *Adding a new node to the group in PBFT*
    - Need new message(s), like JOIN node
    - Need to inform every node (replicas/clients) of the secret key for the new joining node:
        - JOIN must be sent to all existing replicas
        - JOIN also must be sent to all clients
    - Catch up new node: deliver the latest checkpoint to the node. Easiest: every node sends latest checkpoint to new node.
    - The current primary also needs to re-send all the pre-prepares to new node

PBFT client’s view

- Client sends a msg <REQ, o, t, c>_c
- o : operation, t : timestamp at client, c : client
- t provides *exactly once* (RPC) semantics : services executes o once (not 0 times, and not multiple times), it discards any duplicates of REQ — does not execute them, and delivers previous output for o
- Client receives <REPLY, v, t, c, i, r>_i
- v : view number, t : timestamp (of matching req), c : client, i : replica id, r : output / result.
- Client waits for f+1 REPLY msgs (from different replicas, sam t and same r)
    - At most f byz => *at least 1* in f+1 non-faulty => sufficient to detect disagreement (among REPLYs)
    - Assuming that there is underlying agreement protocol that guarantees correct replicas agree with each other

PBFT pprepares

- Pre-prepare, prepare, commit phases (kind of like 3-phase commit)
- <<PRE-P, v, **n**, d>_p, m> to all replicas/backups
- v is the view, n, is sequence number, **d** = D(m) digest of m. (This is the only time we deliver m to all the replicas: use different transport to do so). The d allows us to “talk about” m without including m.
- If backup I accepts the PRE-P, it enters *prepare* phase, and multicasts <PREP, v, **n**, d, i>_i to all nodes
- Together, these guarantee total order for the request *within a view*
- A replica is prepared if received 2f prepares from different backups (2f+1) total: 2f+1 for slot n and msgs m, and another 2f+1 for slot n and msg m’, will overlap in f+1 nodes => 1 honest node in overlap => 1 honest node inconsistent => contradiction.

PBFT commit

- Once *prepared*, replicas send <COMMIT, v,n,D(m),i>_i
- *committed and committed-local*
- committed-local => committed is true , and then **execute m**, to generate result **r**
- Committed-local true if I has accepted 2f+1 commits (including itself); same rationale as before regarding at least one honest replica in overlap
- Commit protocol: ordering across views
- Each replica then just send Client receives <REPLY, v, t, c, i, **r**>_i to the original client.
- The commit part of PBFT is a bit like 2 phase commit at the core
- *Looong proof in a technical report (that needs to consider all corner cases/inter-leavings of events)*

PBFT

- The ugly parts
- They haven't built the view change protocol! No evaluation of corner cases/failures/ Checkpoints are not evaluated.
- The delay accommodation monotonically increases by 2x without ever halfing
- Evaluation is on NFS benchmark (no real clients)
- Number of nodes is … 4 in eval (f = 1); the most optimistic use-case of their system (in terms of performance, and failure assumption)
- Multicast overhead becomes less efficient with more nodes. Message overhead (bandw) not evaluated.
- They only evaluated latency of operations (but esp. for file system… bandw is critical!)
- No comparison with existing system (maybe not so ugly, considering what existed at the time)
- PBFT-NSF FS state kept in memory.. compared against disk-based NSF server!?

Next: RPC

- Paper will be much simpler (yay)
- Focus on LPC -> RPC idea
- Particularly, semantics of RPC (transparency that you can/cannot achieve)