Practical Byzantine Fault Tolerance

PWL #8 @ ATL

Samrit Sangal

The problem

A node that is faulty can exhibit arbitrary behavior. No assumptions can be made about the nature of the failures.

Transport is assumed to be unreliable. May have arbitrary delays, deliver messages out of order, duplicate messages or fail to deliver messages all together.

The solution should stay available but still remain performant

Significance of this paper

- Most work before that assumed benign faults or assumed synchrony (time bounds on message delivery) for safety.
- This was the first to address the presence of malicious actors
- This was the first algorithm that proposed a performant asynchronous state-machine replication protocol.
- Guarantees safety and liveness, assuming some upper bound on the number of faulty nodes
- Recent interest in the blockchain community. This addresses the core problem.

Agenda

- Assumptions and Terminology
- Protocol
- Optimizations
- Implementation and Performance
- Limitations
- Current related work

Assumptions

- In a network of n nodes where n = 3f+1, we can have at most f faulty nodes.
- The state machine being replicated is deterministic. To guarantee safety, it is enough to agree on the ordering of messages.
- Cryptographic algorithms for signing and hashing are secure and correct.
- There is some diversity in the ownership of the nodes. They are not all controlled by the same entity.

Terminology

Safety: Ability of a distributed system to guarantee state. The service should behave like one centralized service that never failed.

Liveness: The clients eventually receive a response to their requests. (Does rely on synchrony for liveness)

Terminology

All replicas are numbered.

View: Numbered configuration of the nodes. The system moves through consecutively numbered views. In a given view, the primary replica is

$$p = v \bmod |R|$$

Where

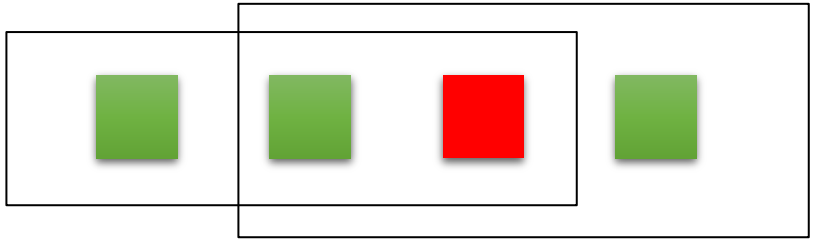
v = view number

|R| = number of replicas

All other nodes are called backups.

Terminology

Quorums: Sets of at least 2f + 1 replicas. They intersect in at least one correct replica.

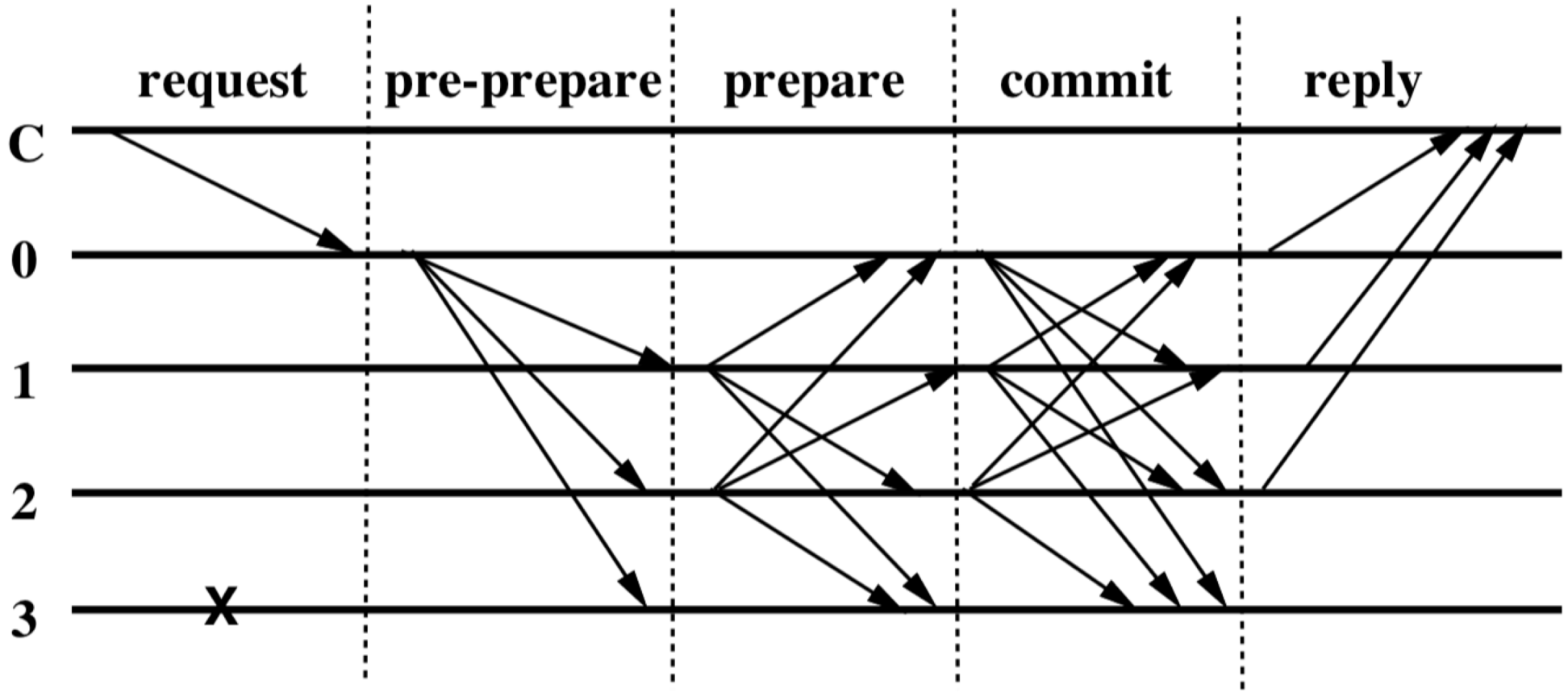


Certificates: These are a set of signed messages from a quorum. These serve as a proof of consensus and also as a check on the primary.

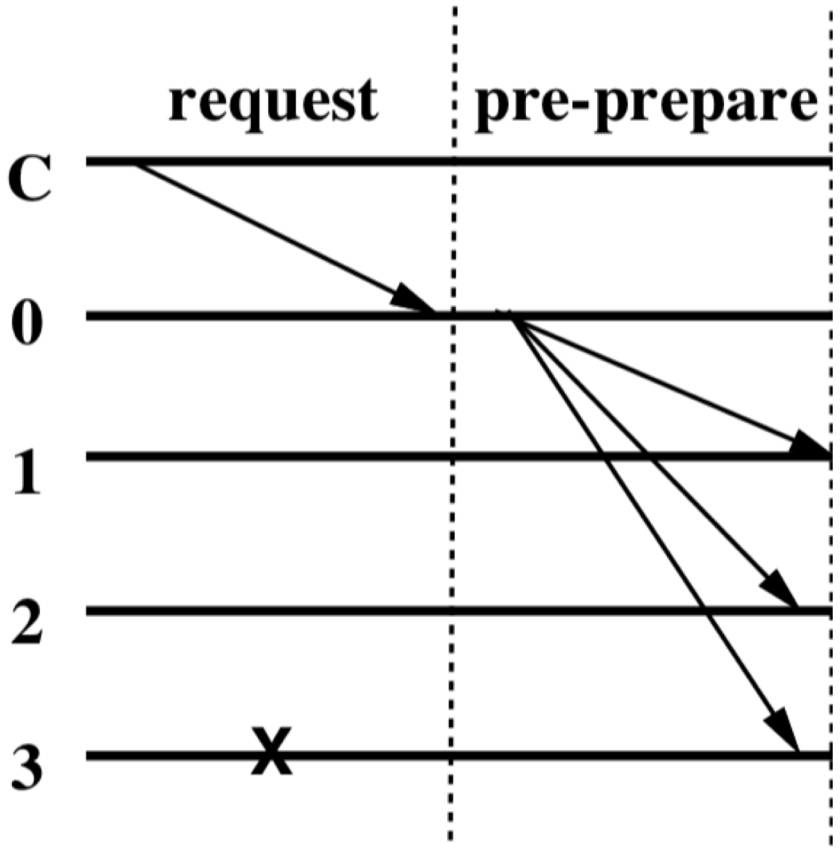
Algorithm

- Client sends a request to invoke an operation to the primary
- The primary multicasts the request to backups
- Three phase commit
- Replicas execute the request and send a reply directly to the client
- Client waits for f+1 replies from different replicas with the same result.

Normal – Case Operation

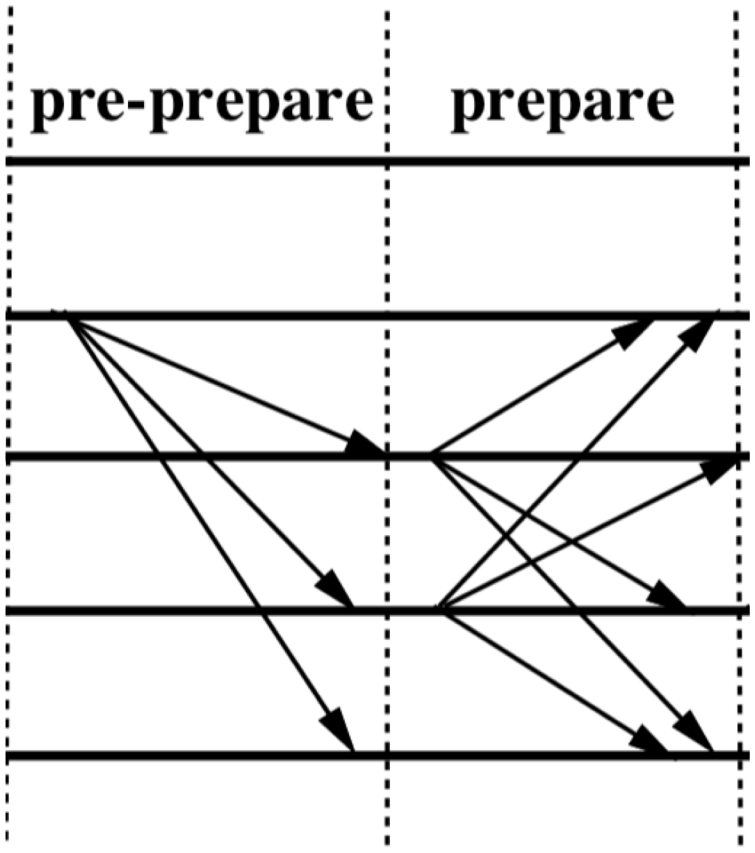


Normal-Case Operation (pre-prepare)



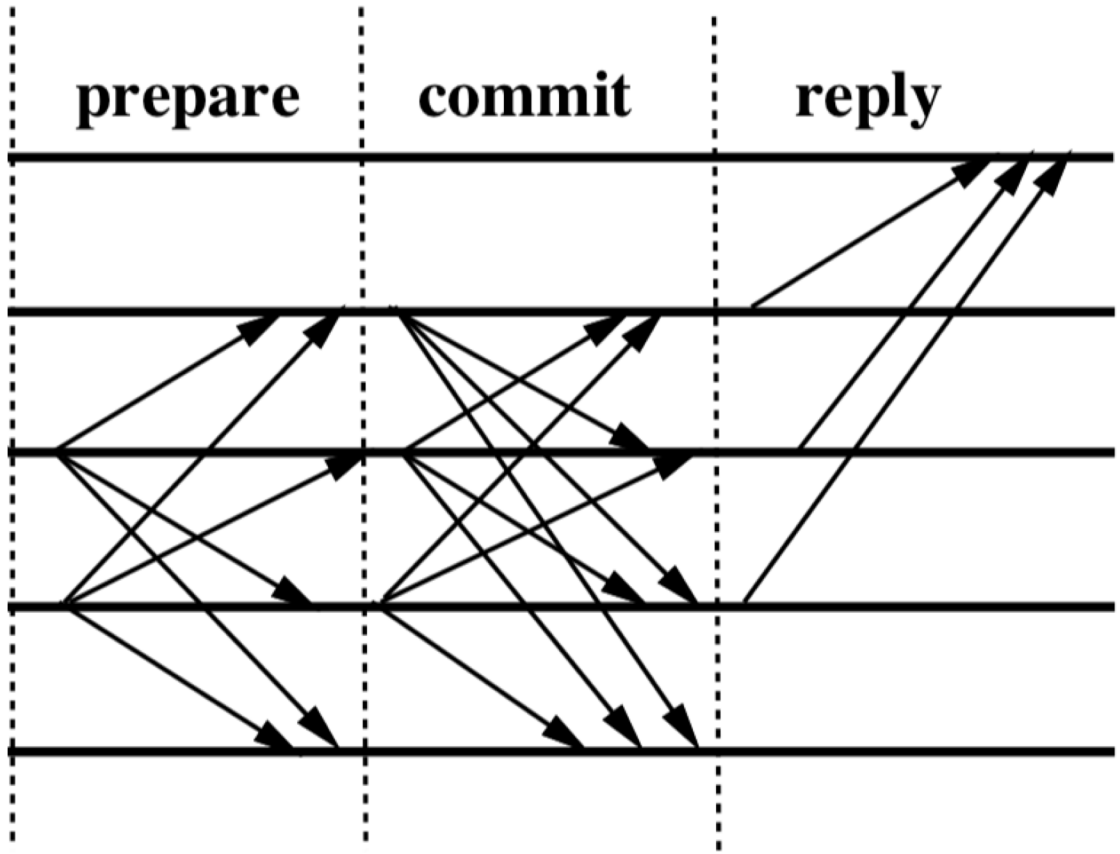
- Primary assigns a sequence number
- Primary sends message m,
  m = ((PRE-PREPARE, v, n, d)$_\sigma$, m)
  where,
  - v = view number
  - n = request sequence
  - d = m digest
  - σ = Digital signature of primary
- Backups will accept this message if
  - they are in view v
  - d is the digest for m
  - signatures are correct
  - they have never accepted a message with n in v.
  - sequence number is within pre-decided bounds

Normal-Case Operation (prepare)



- Send prepare message m to all replicas where
  $m = (PREPARE, v, n, d, i)_{\sigma}$
  where,
  v = view number
  n = request sequence
  d = m digest
  σ = Digital signature of replica
- A replica accepts this prepare message if
  - signatures are correct,
  - views match
  - n is within bounds
- prepared(m, v, n, i) = true if, i has in its log,
  - request m,
  - pre-prepare for m in view v with sequence n
  - 2f matching prepares from different backups

Normal-Case Operation (commit)



- Multicast commit message m, where m = (COMMIT, v, n, D(m), i) $_{\sigma}$
- Replicas accept commits if
  - Properly signed
  - they are in the same view
  - sequence number is within bounds
- Must wait for 2f commits before executing
- Clients accept a result if they get the same result from f+1 different replicas
- Guarantees ordering across views

Garbage Collection

- Mechanism to discard messages from the log
- Can only be discarded if at least f + 1 non-faulty replicas have executed the request, and can provide proof to back this claim up.
- A stable checkpoint is a check point with proof.
- A replica can start a checkpoint when it receives a message with n mod 100 = 0, by sending checkpoint message m, where
  m = (CHECKPOINT, n, d, i) $_{\sigma}$
  where,
  n = sequence number of the last request executed
  d = digest of the state after n was executed
- The usual, wait for quorum, and store this certificate with the checkpoint.
- All earlier checkpoints and log messages are discarded.
- Update bounds for sequence number using some pre decided constant

View-Change

- For liveness, backups have a timer, if a request doesn't complete, or primary behaves arbitrarily, start a view change by sending message m, where
  m = (VIEW_CHANGE, v+1, n, C, P, i)$_\sigma$
  where,
  n = Sequence number of last stable checkpoint
  C = Certificate that proves correctness of last stable checkpoint
  P = p-certificate along with matching pre-prepare messages for all requests greater than n

New-View

- When the primary for v+1 receives 2f VIEW-CHANGE requests, it multicasts a message m where
  m = (NEW-VIEW, v+1, V, O)$_\sigma$
  where
  V = All the valid view change messages received
  O = Set of pre-prepare messages for view v+1
- If no messages are pending after the last stable checkpoint a pre-prepare for a NO-OP request is sent
- Backups accept the new-view if
  - signatures are correct
  - V is correct,
  - O is correct

Optimizations

- Only one replica send the result, rest only send digests
- Tentative replies results in 4 messaging rounds instead of 5, 2f+1 tentative replies guarantee correctness.
- Read-only operations, requests that execute immediately. Client must wait for 2f+1 similar replies before accepting the results of a read-only operation.
- Digital signatures are expensive, replace with MACs, which are three orders of magnitude faster to compute. Each channel has a shared key. MAC is computed by hashing the message+secret key and taking the 10 LSBs of this 16 byte value.

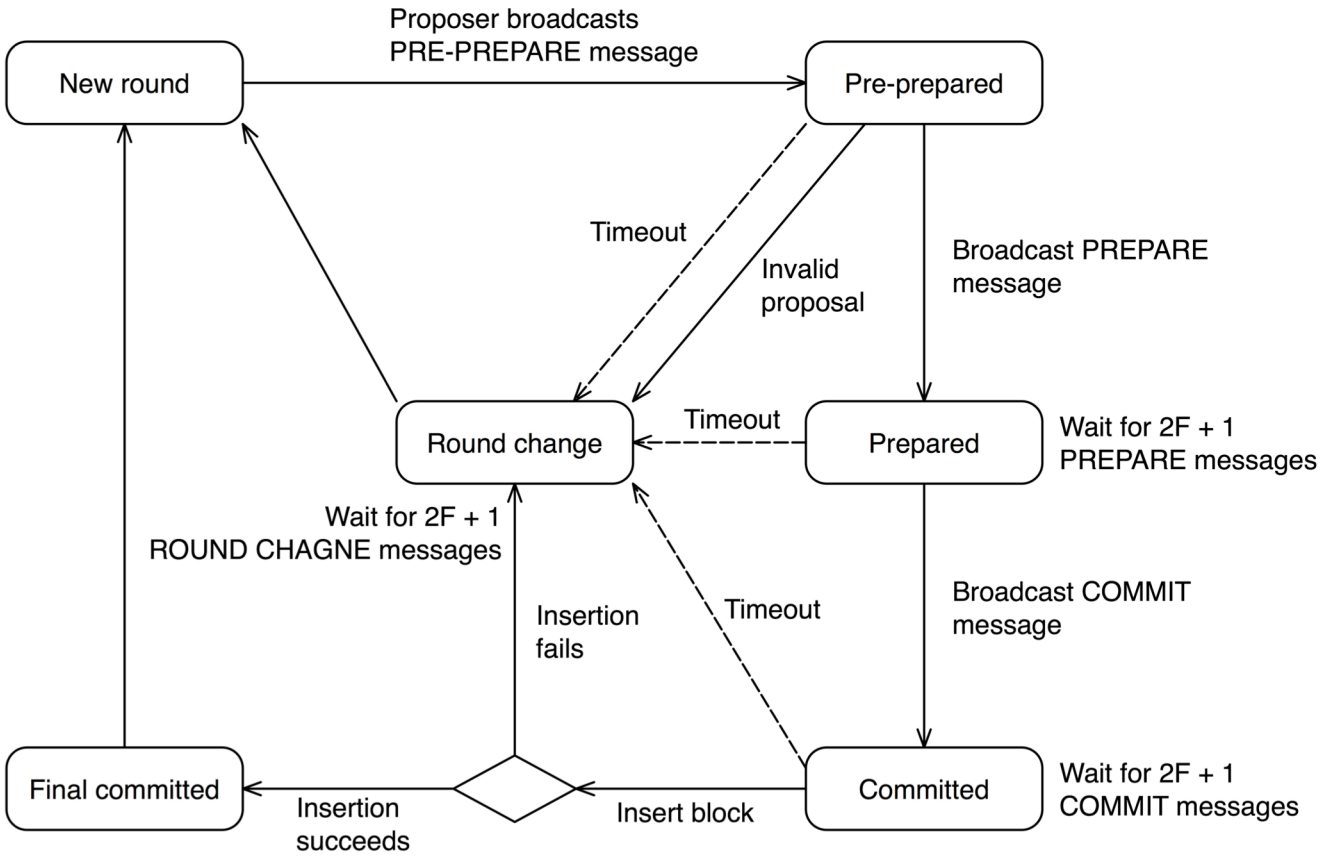
Performance and Implementation

- Created an open source abstraction library, implementing a network file system
- Uses copy-on-write optimizations and incremental digests
- Measures performance using the Andrew benchmark and a micro benchmark. The tests show that PBFT only about 3% slower than NFS.

Limitations

- Extremely chatty, scales poorly for large networks
- Paper doesn’t really go into any discussion about recovering from faults. Certain optimizations impede recovery (MACs) even though they improve performance.
- Not much to do about Byzantine clients accept authenticate and authorize
- Its quite complicated to implement in practice.

Current Related Work



Istanbul BFT – variation of pbft for consensus in Ethereum.

Thank you!