Practical Byzantine Fault Tolerance

Castro and Liskov
SOSP 99

Why this paper?

- Kind of incredible that it’s even possible
- Let alone a practical NFS implementation with it

- So far we’ve only considered fail-stop model

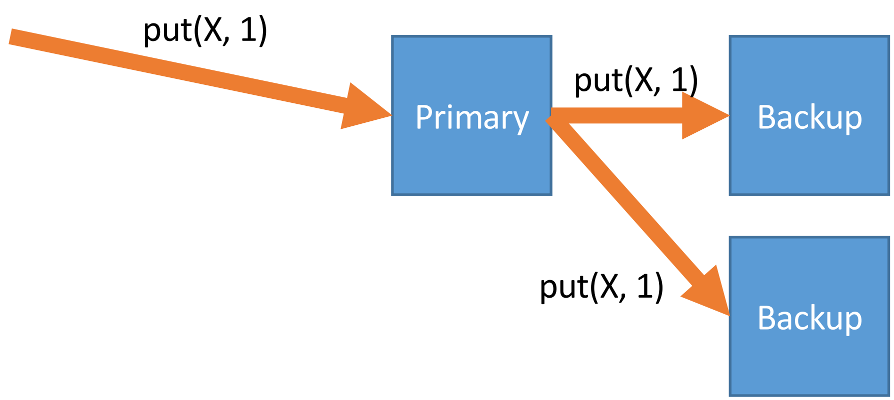
- Quite a bit of research in this area
- Much less real-world deployment
- Most systems being built today don’t span trust domains
- Hard to reason about benefits on compromise

What is Byzantine Behavior?

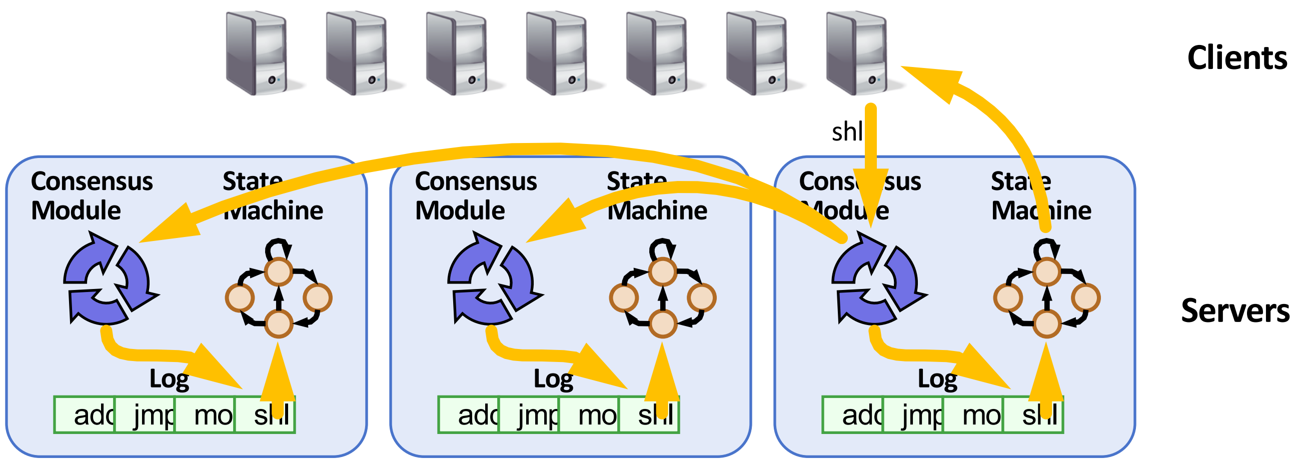
- Anything that doesn't follow our protocol.
- Malicious code/nodes.
- Buggy code.
- Fault networks that deliver corrupted packets.
- Disks that corrupt, duplicate, lose, or fabricate data.
- Nodes impersonating others.
- Joining cluster without permission.
- Operating when they shouldn't (e.g. unexpected clock drift).
  - Service ops on a partition after partition was given to another
- Really wicked bad stuff: any arbitrary behavior.
- Subject to restriction: independence; will come back to this.

Review: Primary/Backup

- Want linearizable semantics
- f + 1 replicas to tolerate f failures
- Runs into problems when “view changes” are needed (Lab 2).



Review: Consensus



- Replicated log => replicated state machine
  - All execute same commands in same order
- Consensus module ensures proper log replication
- Makes progress if any majority of servers are up
  - 2f + 1 servers to remain available with up to f failures
- Failure model: fail-stop (not **Byzantine**), delayed/lost messages

3f + 1?

- At f + 1 we can tolerate f failures and hold on to data.
- At 2f + 1 we can tolerate f failures and remain available.
- What do we get for 3f + 1?
- SMR that can tolerate f malicious or arbitrarily nasty failures

First, a Few Issues

1. Caveat: Independence
2. Spoofing/Authentication

The Caveat: Independence

- Assumes independent node failures for BFT!
- *Is this a big assumption?*
- We actually had this assumption with consensus
  - If nodes fail in a correlated way it amplifies the loss of a single node
  - If factor is > f then system still wedges.
- Put another way: for Paxos to remain available when software bugs can produce temporally related crashes what do we need?
  - 2f + 1 independent implementations…

The Struggle for Independence

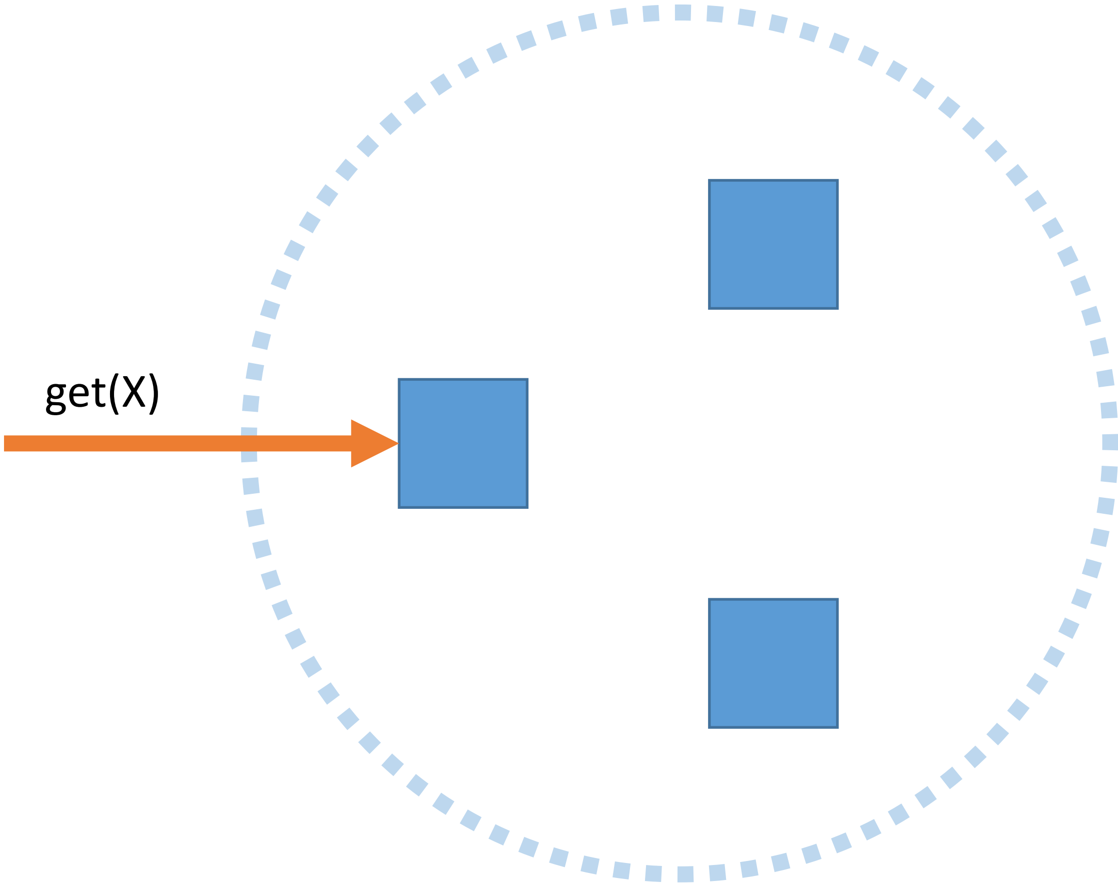
- Same here: for true independence we'll need 3f + 1 implementations
- But it is more important here

1. Nodes may be actively malicious and that should be ok.
   - But they are looking for our weak spot and will exploit to amplify their effect.

2. If > f failures here *anything* can happen to the data.
   - Attacker might change it, delete it, etc… We'll never know.

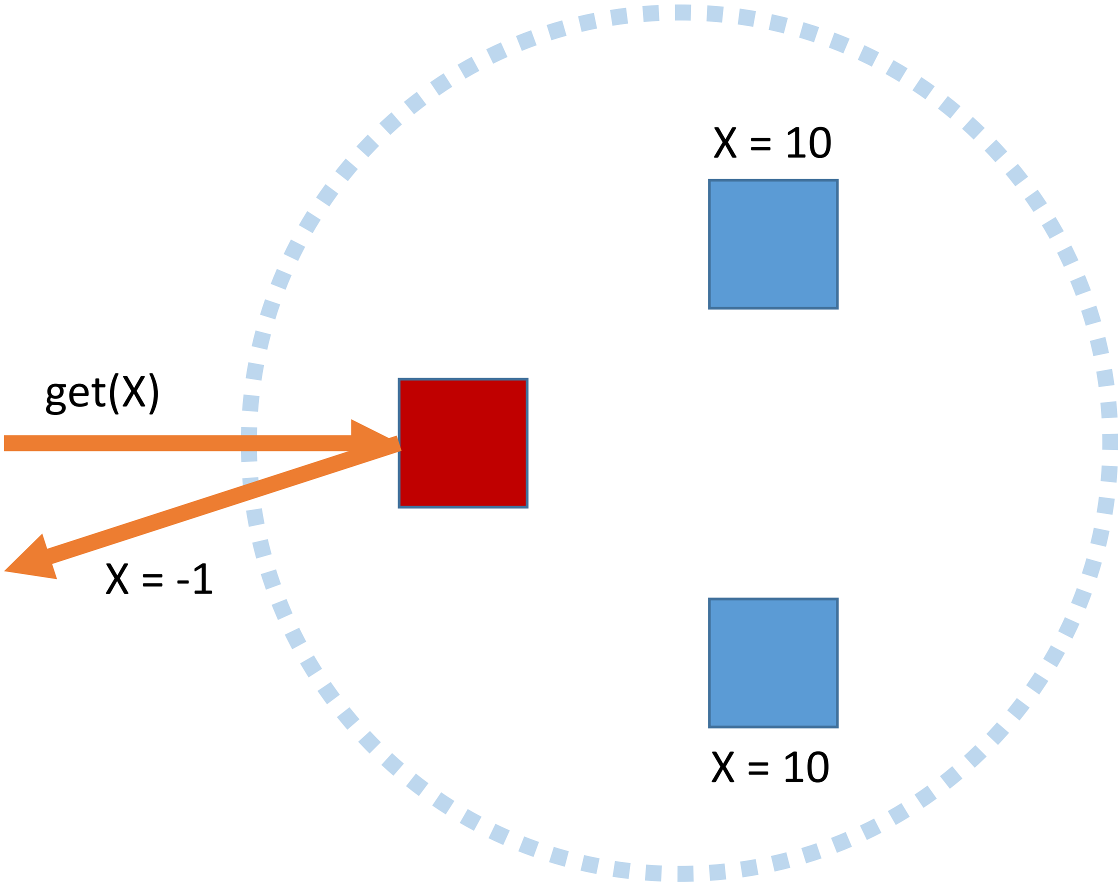
- Requires different implementations, operating systems, root passwords, administrators. Ugh!

Spoofing/Authentication



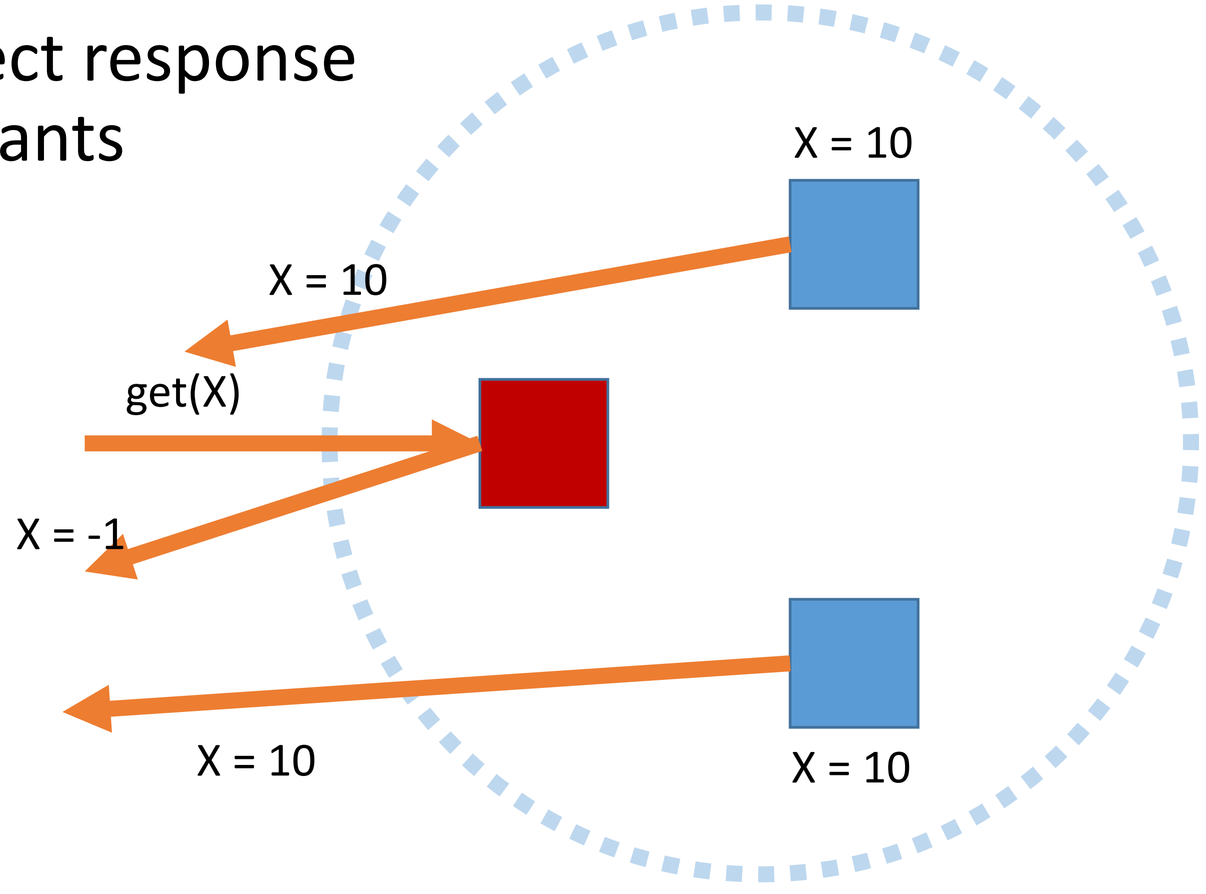
Malicious Primary?

- Might lie!



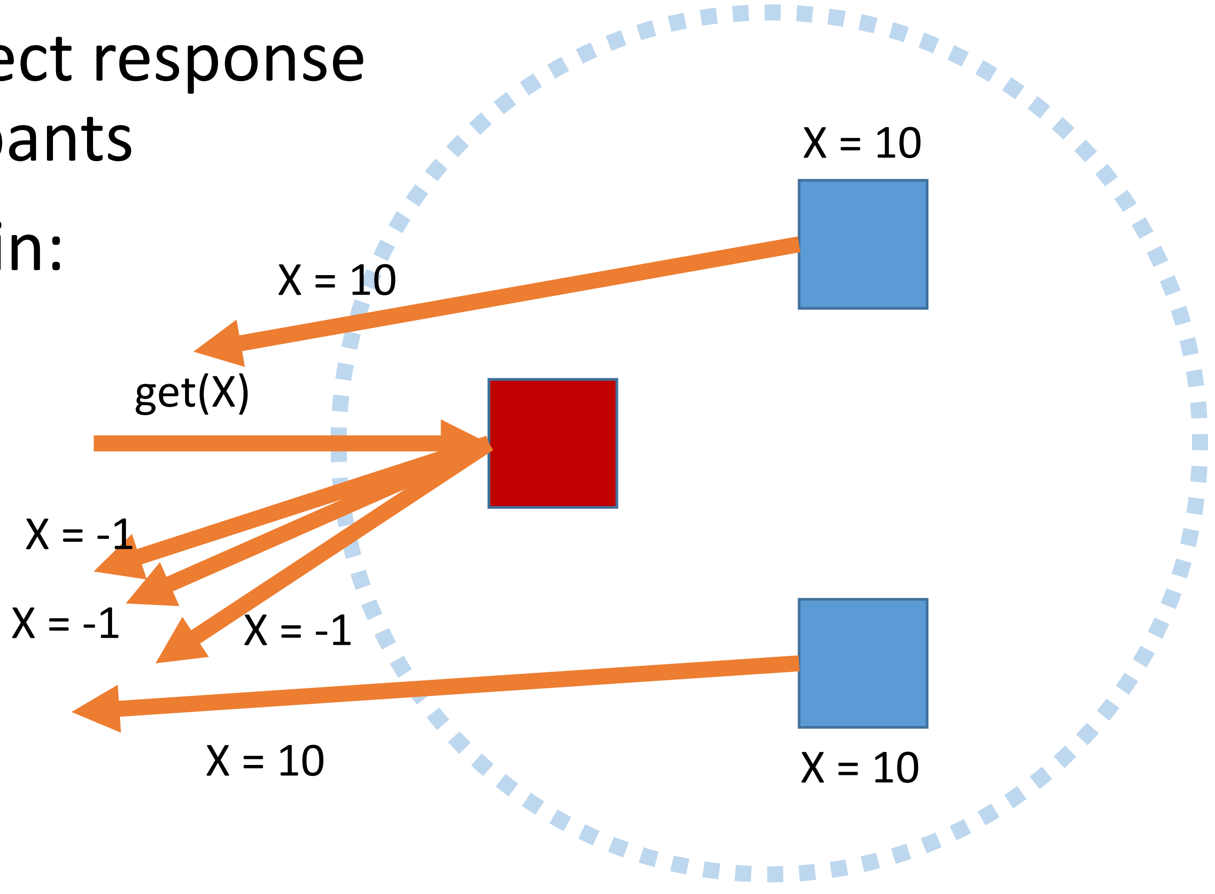
Malicious Primary?

- Might lie!
- Solution: direct response from participants



Malicious Primary?

- Might lie!
- Solution: direct response from participants
- Problem again: primary just lies more

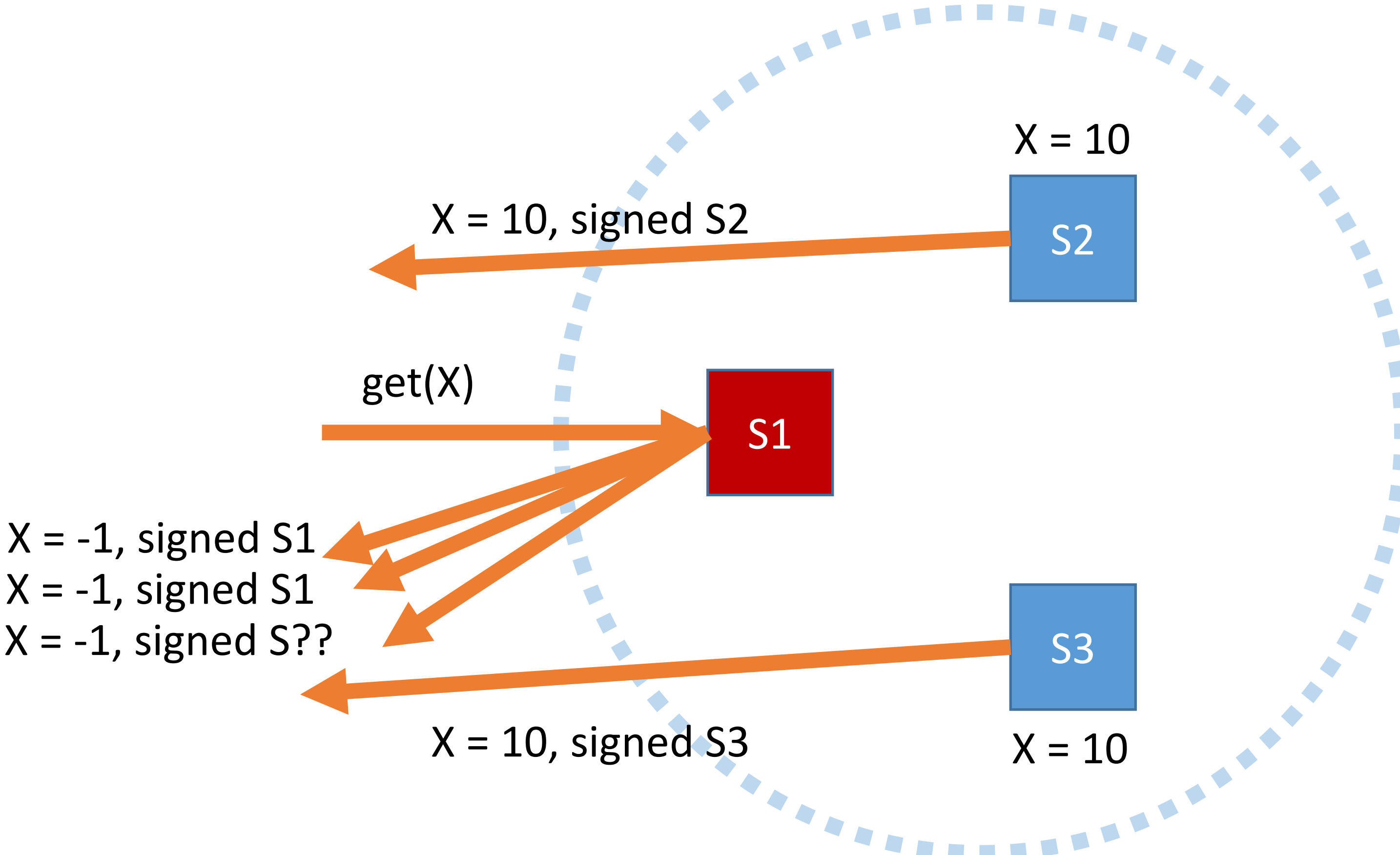


The Need for Crypto

- Need to be able to authenticate messages
- Public-key crypto for signatures
- Each client and server has a private and public key
- All hosts know all public keys
- Signed messages are signed with private key
- Public key can verify that message came from host with the private key
- While we’re on it: we’ll need hashes/digests also

Authenticated Messages

- Client rejects duplicates or unknown signatures



How is this possible? Why 3f + 1?

- First, remember the rules
- Must be able to make progress with n minus f responses
  - $n = 3f + 1$
  - Progress with $3f + 1 - f = 2f + 1$
  - Often 4 total, progress with 3
- Why? In case those f will never respond

Try 2f + 1, f = 1

- Goal: make (safe) progress with only 2 of 3 responses.

Try 2f + 1, f = 1

- Problem: what if S3 wasn't down, but slow
- Instead the failure is a compromised S2
- Client can wait for f + 1 matching responses

Try 2f + 1, f = 1

- Problem: what if S3 is behind, doesn't know value of X yet?
- Can't distinguish truth without f + 1 known good values
- Fix: replicate to at least 2f+1, tolerate f slow/down => 3f+1
- 2f + 1 - f = f + 1, enough to determine truth in face of f lies

3f + 1

- Progress with only 2f + 1 responses and safe
- Among 2f + 1 only f can be bogus. f + 1 > f.

10,000ft View

1. Client sends request to primary.
2. Primary sends request to all backups.
3. Replicas execute the request and send the reply to the client.
4. Client waits for $f + 1$ responses with the same result.

Protocol Pieces

- Deal with failure of primaries
  - View changes (Lab 2/4 style)
  - Similar to Raft, VR
- Must order operations within a view
- Must ensure operations execute within their view

Views

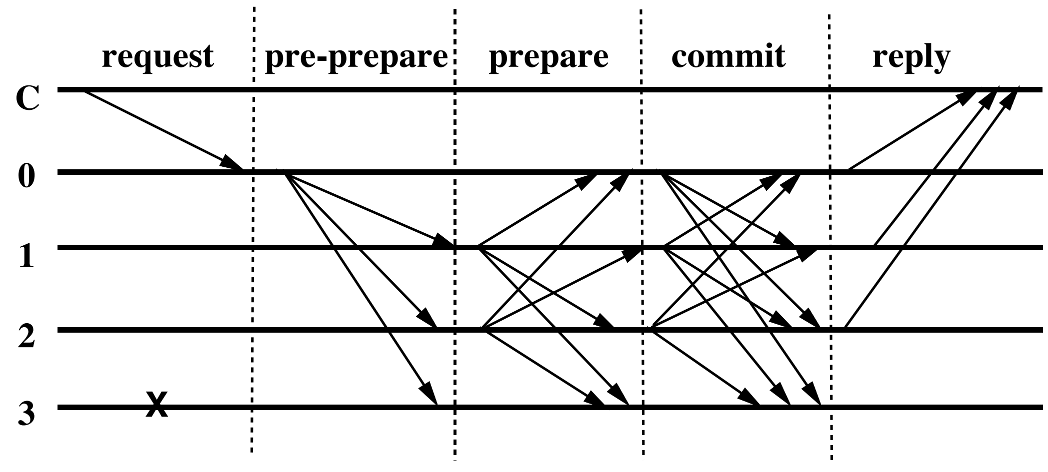
- System goes through a series of views
- In view v, replica (v mod (3f+1)) is designated primary
  - Responsible for selecting the order of operations
  - Assigns an increasing sequence number to each operation
- Tentative order subject to replicas accepting
  - May get rejected if a new view is established
  - Or if order is inconsistent with prior operations

Request Handling Phases

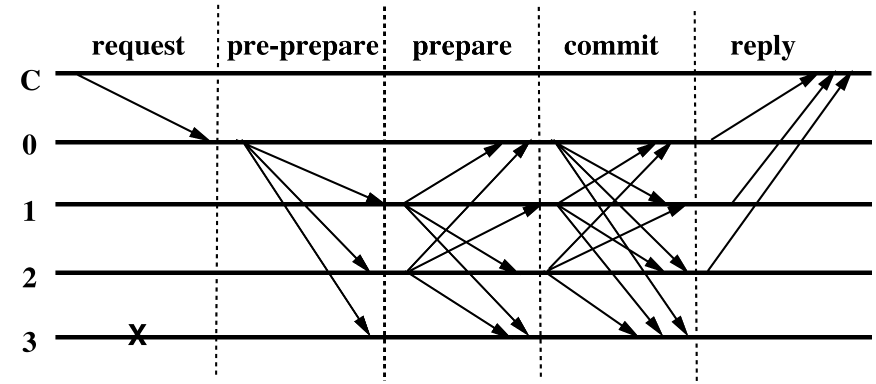
- In normal-case operation, use two-phase protocol for request r:
- Phase 1 (pre-prepare, prepare) goal:
  - Ensure at least f+1 honest replicas agree that
    If request r executes in view v, will execute with seqn
- Phase 2 (prepare, commit) goal:
  - Ensure at least f+1 honest replicas agree that
    Request r has executed in view v with seqn
- 2PC-like:
  - Phase 1 quibble about order, Phase 2 atomicity

Phase 1

- Client to Primary
  {REQUEST, op, timestamp, clientId}$_{sc}$
- Primary to Replicas
  {PRE-PREPARE, view, seqn, h(req)}$_{sp}$, req
- Replicas to Replicas
  {PREPARE, view, seqn, h(req), replicaId}$_{sri}$

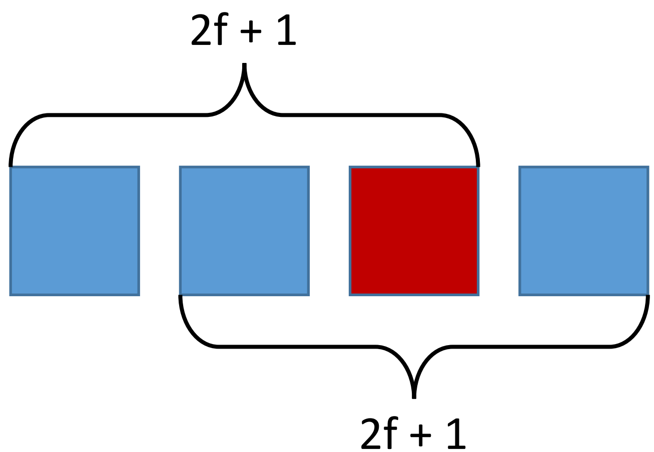


Phase 1



- Each replica waits for PRE-PREPARE + 2f matching PREPARE messages
- Puts these messages in its log
- Then we say prepared(req, v, n, i) is TRUE
- If prepared(req, v, n, i) is TRUE for honest replica r_i then prepared(req', v, n, j) where req' != req FALSE for any honest r_j
  - So no other operation can execute with view v sequence number n

Why No Double Prepares?



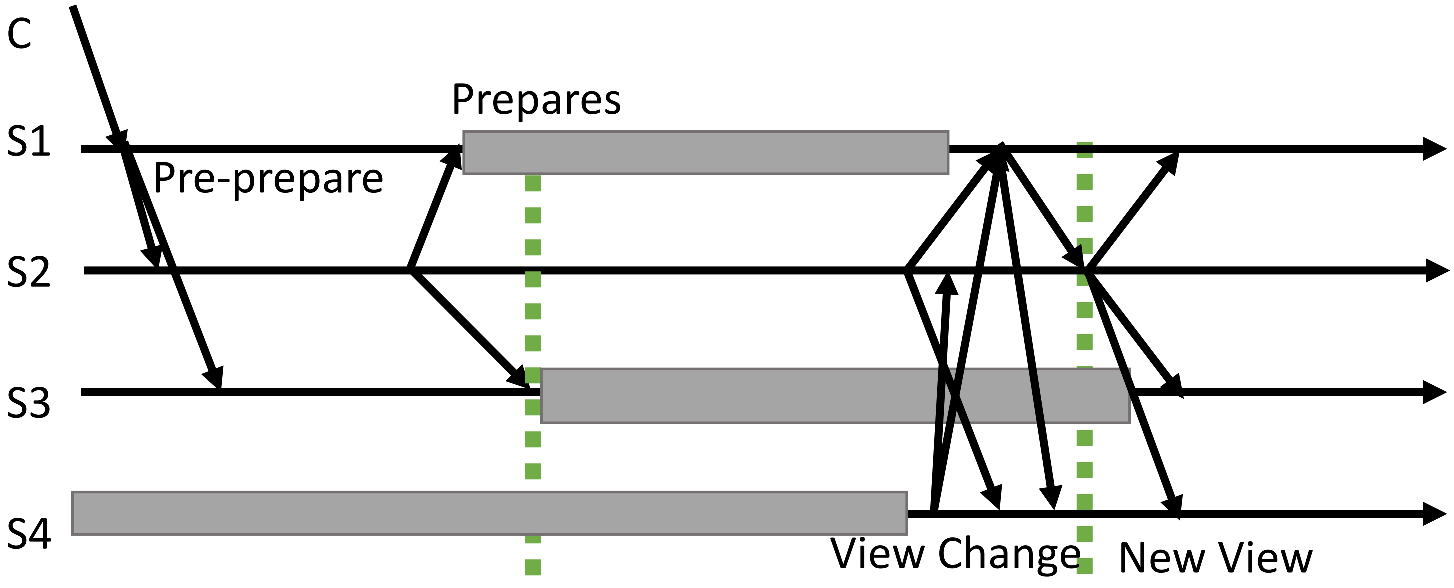
prepared(req, v, n, i) → not prepared(req', v, n, j) for honest r_i and r_j

Honest intersection of maximally disjoint 2f+1 sets is non-empty

Phase 2

- Problem: Just because some other req' won't execute at (v, n) doesn't mean req will

Problem: Prepared != Committed



- S3 prepared, but couldn't get PREPARE out
- S2 becomes primary in new view
- Can't find PRE-PREPARE + 2f PREPAREs in any log
  - S1: {S1, S2}, S2: {S1, S2}, S4: {}
- New primary must fill 'hole' so log can move forward

Phase 2

- Make sure op doesn't execute until prepared(req, v, n, i) is TRUE for f+1 non-faulty replicas
- We say committed(req, v, n) is TRUE when this property holds
- How does replica know committed(req, v, n) holds?
- Add one more message: r_i -> R
  {COMMIT, view, seqno, h(req), replicaId}
- Once 2f+1 COMMITs at a node, then apply op and respond to client

View Changes

- Allows progress if primary fails (or is slow)
- If operation on backup pending for long time
{VIEW-CHANGE, view + 1, seqn, ChkPointMgs, P, i}$_{si}$
- New primary issues NEW-VIEW once 2f VC msgs
  - Includes signed VIEW-CHANGEs as proof it can change view
  - Q: What goes wrong without this?
- Then, for each seqno since lowest stable checkpoint
  - Use P from above: set of sets of PRE-PREPARE + 2f PREPARES
  - For seqno with valid PRE-PREPARE + 2f PREPARE, reissue PRE-PREPARE in v + 1
  - For seqno not in P, {PRE-PREPARE, v + 1, seqno, null}

- Once committed at least f + 1 non-faulty replicas have agreed on the operation and its placement in the total order of operations
    - Even across view changes

Checkpoints/GC

- Need to occasionally snapshot SM and truncate log
- Problem: how can one replica trust the checkpoint of another?
- Idea: at (seqn mod 100) broadcast $\{\text{CHECKPOINT, seqn, h(state), i}\}_{si}$
- Once 2f+1 CHECKPOINTs have been collected then can trust CHECKPOINT at seqn with correct digest (at least f + 1 non-faulty servers have a correct checkpoint at seqn)

Liveness – View Changes

- Interesting issue: can't let a single node start a view change!
- Why? Could livelock the system by spamming view changes.
- Resolution: wait for f + 1 servers to timeout and independently send VIEW-CHANGE requests.
- Interacts with an optimization: to try to ensure that view changes succeed if any node that gets more than f + 1 VIEW-CHANGE requests issues one as well.
  - This prevents cases where they timeout slowly and then the oldest VIEW-CHANGE issuer rolls over to VIEW-CHANGE v + 2.
  - Have to be careful still: need to wait on this optimization until f + 1 VIEW-CHANGES away from v.
  - Why? Otherwise might be doing the bidding of a malicious node.

Discussion

- What problem does this solve?
- Would your boss be ok with 4 designs/implementations?
- How can system tolerate more than f (non-simultaneous) failures over its lifetime?
  - Periodically recover each server? Could help some…
  - What if private key compromised?
- Important point: it is possible to operate in the face of Byzantine faults
  - Maybe even efficiently

Performance Tricks

- Don’t have replicas respond with operation results, just digests
  - Only primary has to give result
- Delays: client to primary, pre-prepare, prepare, commit, reply
  - Idea: commit prepared operations tentatively.
  - If wrong, rollback.
  - Operations unlikely to fail to commit if they prepare successfully.
- Tentatively execute reads against tentative operations, but withhold reply until all operations read from have committed.

Crypto

- Can’t afford digital signatures on all messages to authenticate
- Instead all pairs of hosts share a secret key
- Send MAC of each message (h(m + secret key)) to verify integrity, authenticity.
- Problem: what about messages with multiple recipients?
  - e.g. client operation request message?
  - Can’t let faulty nodes spoof operations.
  - Put a vector of MACs in for the message, one for every node in the system.
  - Probably 4 or 7 hosts. Constant time to verify, linear to generate.
  - 37 replicas, MAC vectors still 100x faster to generate than 1024 bit RSA sig.
  - Output is also smaller than a 1024 bit sig.

Why Pre-prepare, Prepare, Commit?

- Pre-prepare
    - Broadcast viewno, seqn, and message digest.
    - Backup accepts
        - If digest is ok for the message
        - Backup is in same view
        - Hasn’t accepted a pre-prepare for seqno in viewno with a different digest.
    - If it accepts it broadcasts prepare
- Prepare
- Commit
    - Similar to our decided; informs everyone of the chosen value
    - Difference: can’t take sender’s word for it, need proof that the cluster agrees.

Phase 2

- Just because some other req' won't execute at (v, n) doesn't mean req will
- Suppose r_i is compromised right after prepared(req, v, n, i)
- Suppose no other replica received r_i's PREPARE
- Suppose f replicas are slow and never even received the PRE-PREPARE
- No other honest replica will know the request prepared!
- Particularly if p fails, request might not get executed!