Byzantine Fault Tolerance

Problem

- We handle a larger class of failures where we have buggy servers that have been modified by an attacker / don't follow our protocol
  - Before, we handled only the case where servers just crashed / were network partitioned
- If we allow for f faulty replicas, we need at least $3f + 1$ replicas
- We assume that there are no client failures and that we can't break cryptography
- This paper provides a replicated state machine that can handle these Byzantine faults
- We'll build towards the paper's design

Designs

- We have n servers that are all independent
  - Client and servers have public key pairs such that they can encrypt messages to each other
  - Client sends requests to all servers and waits for all to reply
  - Only proceeds if all n agree
- Problem: if one server is compromised, then one server can stop by disagreeing
- Solution:
  - Replicas vote
  - We have $2f + 1$ servers and client waits for $f + 1$ matching replies
- Problem: $f + 1$ matching replies might be f bad nodes and 1 good
  - The replicas could just reply without actually writing / doing anything
  - The next operation will also wait for $f + 1$ and this might not include the one good node that saw operation 1, so we won't see any disagreement
- New design:
  - $3f + 1$ servers where we wait for $2f + 1$ matching replies
    - This way we must have a majority of the good nodes
  - Now if we have $f, g1, g2, g3$ such that f is faulty
    - First message is replied to by $f, g1, g2, g3$ (write of "A")
    - Second message is replied to by $f, g1, g2$ (write of "B")
    - When we do the read, we will get two that say "A" ($f, g3$ because f is faulty)
      - Two other say "B", so the client doesn't know what to believe but it does not that there is a problem
      - We'll get into repair later
- Problem: To handle multiple clients, we need to process operations in the same order
  - We have a primary pick the order for concurrent client requests
  - We have to worry about a faulty primary now
    - They could:
      - Send wrong result to client
      - Send different ops to different replicas
      - Ignore a client op
    - General approach to dealing with is:
      - Replicas send results directly to client
      - Replicas exchange info about ops sent by primary
      - Clients notify replicas of each operation as well as primary
        - If we make no progress, then we force a change of the primary
    - Replicas can't immediately execute operations it receives from the primary because it might be faulty
      - We need another round of messages to ensure that all good replicas got the same operation

Next Design

- $3f + 1$ servers where one is the primary and f are faulty (primary might be one of those)
  - Client sends request to primary AND to each replica
  - Primary chooses next op and op #
  - Primary sends PRE-PREPARE(op, n) to replicas
    - Each replica sends PREPARE(op, n) to all replicas
  - If replica gets matching PREPARE(op, n) from $2f + 1$ replicas (including itself) and n is the next operation number, then it executes the operation and replies to the client
    - Otherwise it keeps waiting
  - The client is happy when it gets $f + 1$ matching replies
- If the primary is not faulty, the faulty replicas:
  - Cannot forge primary messages since the non-faulty ones know who is the primary and we have cryptography
  - They cannot stop the $2f + 1$ prepares from being sent out
  - The most they can do is just cause delays by DOSing the replicas
- If the primary is faulty:
  - Cannot forge client ops
  - Can't ignore client ops since client sends it to all replicas
  - If the primary sends different ops to different replicas:
    - We have different cases based on how many good nodes received the correct operation
    - If all good nodes get $2f + 1$ matching PREPAREs:
      - Everyone who got it must have gotten the same operation
        - This is because any two sets of $2f + 1$ servers must share a good server
        - Therefore every person who has received $2f + 1$ responses must have a common response that
      - Therefore all good nodes will execute and respond to client and the client is happy
    - If not all but $\geq f + 1$ good nodes get PREPAREs:
      - No disagreement possible and the client receives a response but up to f good nodes don't execute
      - These f remaining nodes can't be used to rollback the op because then there aren't enough nodes with only the old state
        - There would only be these f and then the f faulty
    - If $< f + 1$ good nodes get $2f + 1$ matching PREPAREs:
      - The client will never get a reply since at least $f + 1$ will be stuck waiting for this OP

- How do we know when there is a problem and we have a faulty primary?
  - A replica asks for a view change after some timeout period
  - We can't just change when a single replica asks because then faulty ones will constantly ask for view changes
- We'll get back to discussing how many replicas must ask later
- To ensure that faulty replicas can't always just make themselves the next primary, we keep a **view number** and the next primary is $v\%n$
  - We then have at most f faulty primaries in a row
- However we might have a problem where the new primary did not execute an operation that did properly receive $2f + 1$ PREPAREs
  - Therefore it might try to overwrite an operation even if it is non-faulty
  - We therefore need to wait to execute an operation until we're sure that a new primary has heard about it
  - We need to add a third phase: COMMIT so that we only execute after commit
- New protocol:
  - Client sends op to primary
  - Primary sends PRE-PREPARE(op, n) to all
  - All send PREPARE(op, n) to all
  - After a replica receives $2f + 1$ matching PREPARE(op, n)

        - Send COMMIT(op, n) to all
    - After a replica receives $2f + 1$ matching COMMIT(op, n)
        - Execute op
- Back to the view change procedure:
    - If a replica thinks the primary is faulty, it sends a VIEW-CHANGE message to all other replicas
        - This contains all of the commands the replica has sent prepared messages for
    - When a replica receives a VIEW-CHANGE message from $2f + 1$ replicas, then it sends a NEW-VIEW message to all other replicas containing all VIEW-CHANGE messages it saw
        - This means that it contains a set of things that the $2f + 1$ replicas have committed
            - This means that this contains a comprehensive list up to everything that was committed in the previous view
                - This follows from again the fact that every set of $2f + 1$ replicas shares 1 good replica in common
            - Because of our commit guarantees, this ensures that this primary is up to date and did not lose anything from the previous view
        - It also has a list of everything that was prepared, which can be used to determine things that should have been committed (i.e. >= 2f + 1)
        - The new leader (if faulty) cannot spoof this because the VIEW-CHANGE messages are signed
    - When a replica receives NEW-VIEW from $2f + 1$ replicas, it updates its state to reflect the new view
    - If a replica does not receive NEW-VIEW within a certain time limit, it initiates another view change protocol

Lecture Question

Suppose that we eliminated the pre-prepare phase from the Practical BFT protocol. Instead, the primary multicasts a PREPARE,v,n,m message to the replicas, and the replicas multicast COMMIT messages and reply to the client as before. What could go wrong with this protocol? Give a short example, e.g., ``The primary sends foo, replicas 1 and 2 reply with bar...''

```
Consider two clients sending m and m' in system with 4
servers (3f+1=4) with a malicious primary in some view v.
The malicious primary sends PREPARE(v, n, d) and PREPARE(v,
n, d') to the replicas, where n is the sequence number, d
and d' are the digests for the messages m and m',
respectively. The replicas will accept both PREPARE's
because the protocol in the paper doesn't check for
duplicate sequence numbers for PREPARE's.

Each replica multicasts COMMIT(v, n, d) and COMMIT(v, n,
d'). The fault primary is assumed to be able to control the
network and can delay some of the COMMIT's such that replica
1 receive 2f+1 COMMIT(v, n, d)'s first and replica 2
receives COMMIT(v, n, d')'s first.  Now replica 1 executes m
in slot n and replica 2 will execute m' in slot n, resulting
in divergence state.

You could modify the broken protocol to be less broken and
check for duplicate sequence numbers in prepare phase, but
that is still not good enough, as your answer to the
question illustrates: only one server executes an operation
in view v for n, the remaining server enter a new view
without S3's support, and the new primary commits another
operation in v+1 for n.
```