Byzantine Fault Tolerance

Arvind Krishnamurthy
University of Washington

Fault Tolerance

- We have so far assumed “fail-stop” failures (e.g., power failures or system crashes)
- In other words, if the server is up, it follows the protocol
- Hard enough:
  - difficult to distinguish between crash vs. network down
  - difficult to deal with network partition

Larger Class of Failures

- Can one handle a larger class of failures?
  - Buggy servers that compute incorrectly rather than stopping
  - Servers that do not follow the protocol
  - Servers that have been modified by an attacker
  - Referred to as Byzantine faults

Model

- Provide a replicated state machine abstraction
- Assume 2f+1 of 3f+1 nodes are non-faulty
    - In other words, one needs 3f+1 replicas to handle f faults
- Asynchronous system, unreliable channels
- Use cryptography (both public-key and secret-key crypto)

General Idea

- Primary-backup plus quorum system
  - Executions are sequences of views
  - Clients send signed commands to primary of current view
  - Primary assigns sequence number to client's command
  - Primary writes sequence number to the "register" implemented by the quorum system defined by all the servers

Attacker's Powers

- Worst case: a single attacker controls the *f* faulty replicas
- Supplies the code that faulty replicas run
- Knows the code the non-faulty replicas are running
- Knows the faulty replicas' crypto keys
- Can read network messages
- Can temporarily force messages to be delayed via DoS

What faults cannot happen?

- No more than f out of 3f+1 replicas can be faulty
- No client failure -- clients can never do anything bad (or rather such behavior can be detected using standard techniques)
- No guessing of crypto keys or breaking of cryptography

- Question: in a Paxos RSM setting, what could the attackers or byzantine nodes do?

What could go wrong?

- Primary could be faulty!
  - Could ignore commands; assign same sequence number to different requests; skip sequence numbers; etc.
  - Can equivocate or lie differently to different nodes
- Backups could be faulty!
  - Could incorrectly store commands forwarded by a correct primary
- Faulty replicas could incorrectly respond to the client!

Example Use Scenario

- Arvind:

  echo A > grade

  echo B > grade

  tell Lequn "the grade file is ready"

- Lequn:

  cat grade

Strawman Design

- let us have replicas vote
- 2f+1 servers, assume no more than f are faulty
- client waits for f+1 matching replies
    - if only f are faulty, and network works eventually, must get them!

- what is wrong with this design?

Issues with Design

- f+1 matching replies might be f bad nodes & 1 good
  - so maybe only one good node got the operation!
  - next operation also waits for f+1
  - might not include that one good node that saw op1
- example: S1 S2 S3 (S1 is bad)
  - everyone hears and replies to write("A")
  - S1 and S2 reply to write("B"), but S3 misses it
    - client can't wait for S3 since it may be the one faulty server
  - S1 and S3 reply to read(), but S2 misses it; read() yields "A"
- result: client tricked into accepting out-of-date state

Improved Design

- 3f+1 servers, of which at most f are faulty
- client waits for 2f+1 matching replies
    - f bad nodes plus a majority of the good nodes
    - so all sets of 2f+1 overlap in at least one good node

- does design 3 have everything we need?

Refined Approach

- let us have a primary to pick order for concurrent client requests
- use a quorum of 2f+1 out of 3f+1 nodes
- have a mechanism to deal with faulty primary
  - 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 no progress, force change of primary

PBFT: Overview

- Normal operation: how the protocol works in the absence of failures; hopefully, the common case
- View changes: how to depose a faulty primary and elect a new one
- Garbage collection: how to reclaim the storage used to keep various certificates

Normal Operation

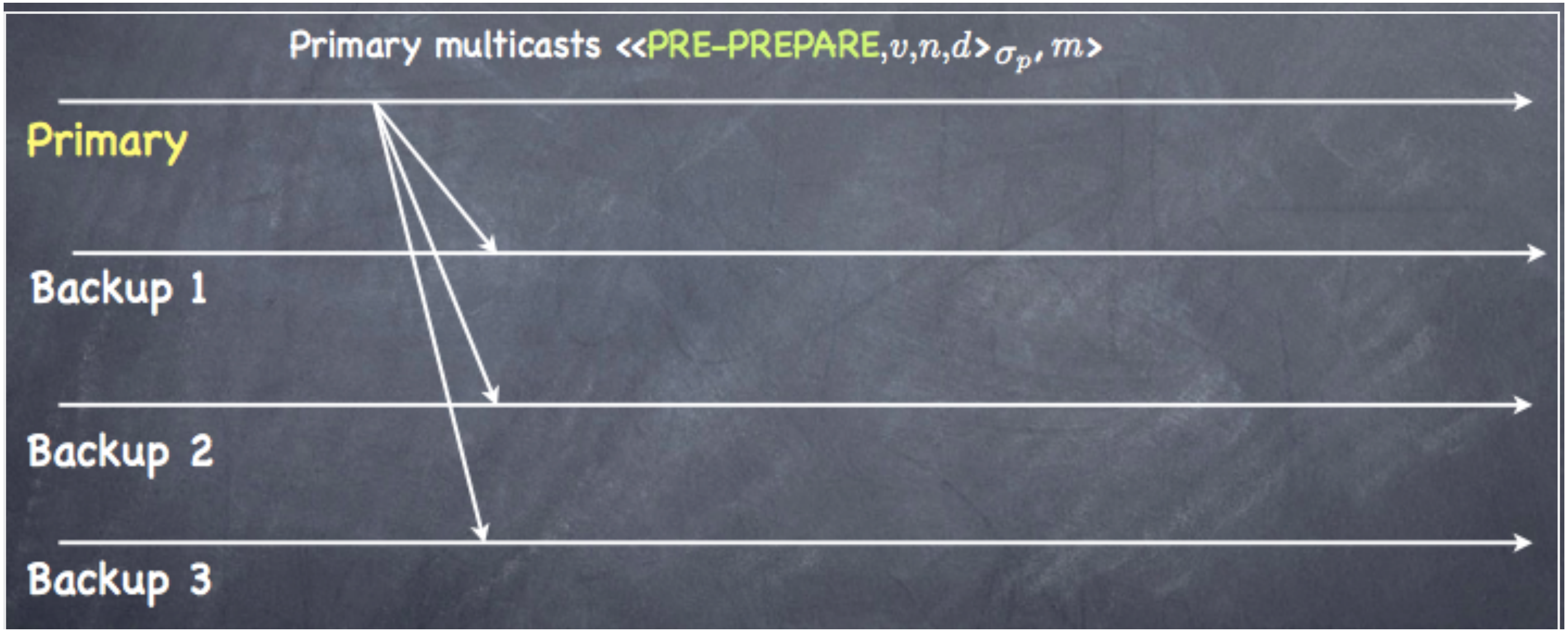
- Pre-prepare: assigns sequence number to request
- Prepare: ensures fault-tolerant consistent ordering of requests within views
- Commit: ensures fault-tolerant consistent ordering of requests across views

- Service state
- Message log with all messages sent/received
- Integer representing the current view number

Client issues request

- o: state machine operation
- t: timestamp
- c: client id

Pre-prepare

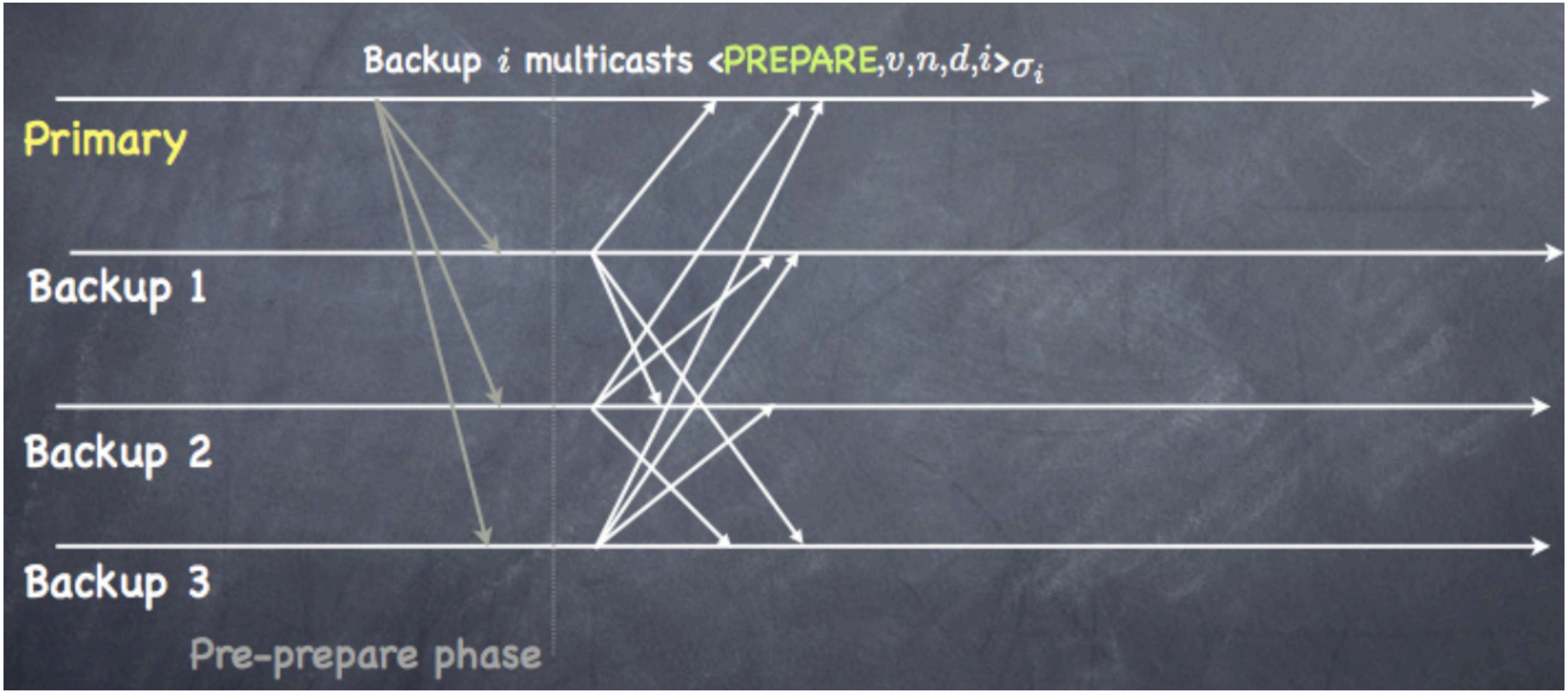


- v: view
- n: sequence number
- d: digest of m
- m: client’s request

Pre-prepare Receipt

- Correct backup accepts pre-prepare if:
  - it is well-formed
  - in the current view
  - it hasn’t accepted a different pre-prepare
  - sequence number is between a low and a high water-mark
- Pre-prepare is logged in a durable log

Prepare



- Correct backup accepts prepare message with usual checks:
  - Well-formed, in current view, between water-marks
  - It is logged in a durable log

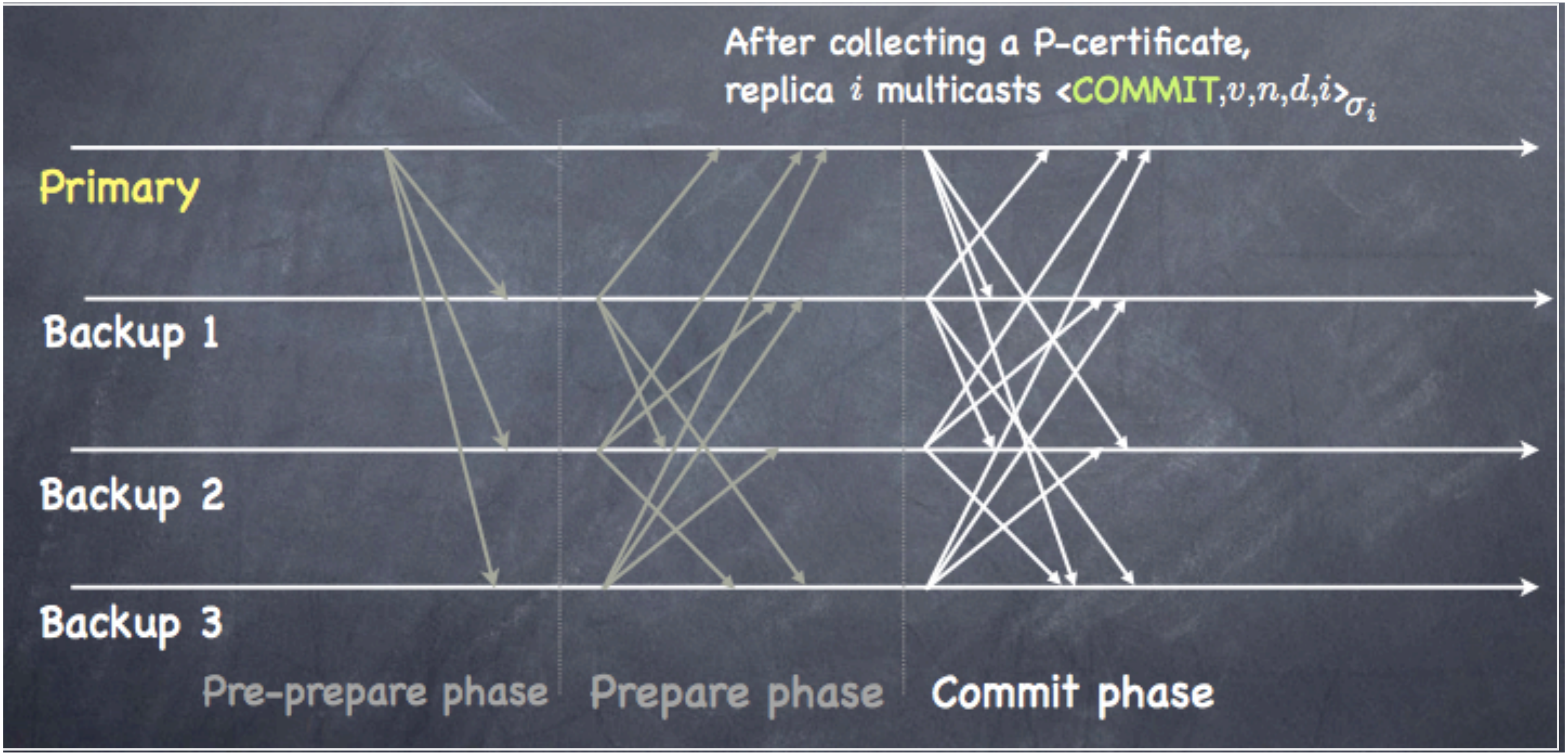
Prepare Certificate

- P-certificates ensure total order within views
- Replica produces P-certificate(m,v,n) iff its log holds:
  - The request m
  - A PRE-PREPARE for m in view v with sequence number n
  - 2f PREPAREs from different backups that match the pre-prepare
- A P-certificate(m,v,n) means that a quorum agrees with assigning sequence number n to m in view v
  - No two non-faulty replicas with P-certificate(m1,v,n) and P-certificate(m2,v,n)

P-certificates are not enough

- A P-certificate proves that a majority of correct replicas has agreed on a sequence number for a client's request
- Yet that order could be modified by a new leader elected in a view change

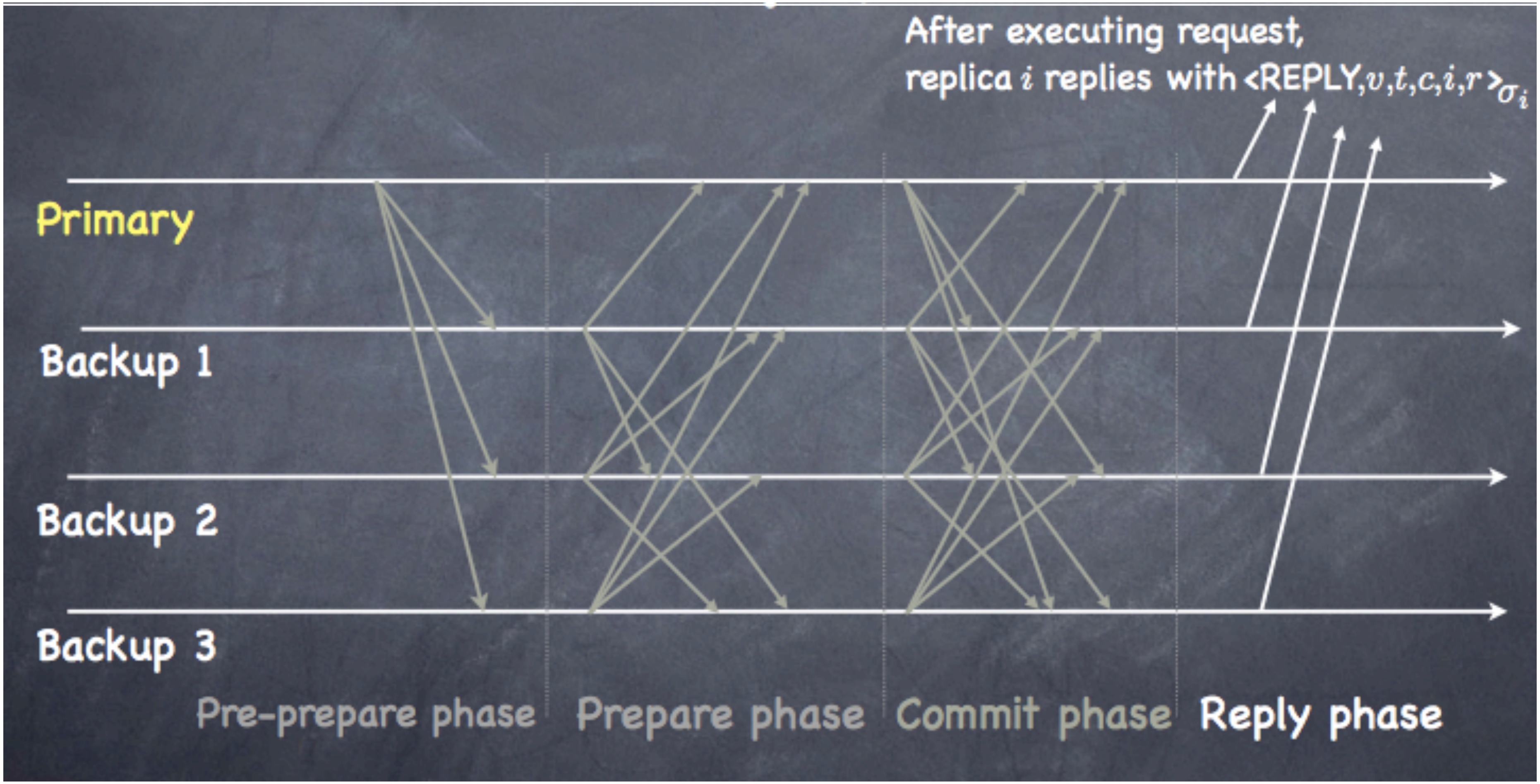
Commit


After collecting a P-certificate,
replica i multicasts <COMMIT,v,n,d,i>$_{\sigma_i}$
Primary
Backup 1
Backup 2
Backup 3
Pre-prepare phase
Prepare phase
Commit phase

Commit Certificate

- C-certificates ensure total order across views
  - can’t miss P-certificate during a view change
- A replica has a C-certificate(m,v,n) if:
  - it had a P-certificate(m,v,n)
  - log contains 2f +1 matching COMMIT from different replicas (including itself)
- Replica executes a request after it gets a C-certificate for it, and has cleared all requests with smaller sequence numbers

Reply



After executing request,
replica i replies with $\langle$REPLY,$v,t,c,i,r\rangle_{\sigma_i}$
Primary
Backup 1
Backup 2
Backup 3
Pre-prepare phase
Prepare phase
Commit phase
Reply phase

Common Case Analysis

- How does this compare to normal Paxos?
- What are missing loose ends in getting this to work?

Backups Displace Primary

- A disgruntled backup mutinies:
  - stops accepting messages (but for VIEW-CHANGE & NEW-VIEW)
  - multicasts <VIEW-CHANGE,v+1, P>
  - P contains all P-Certificates known to replica i
  - A backup joins mutiny after seeing f+1 distinct VIEW-CHANGE messages
- Mutiny succeeds if new primary collects a *new-view certificate* V, indicating support from 2f +1 distinct replicas (including itself)

View Change: New Primary

- The “primary elect” p’ (replica v+1 mod N) extracts from the new-view certificate V :
  - the highest sequence number h of any message for which V contains a P-certificate
  - two sets O and N:
    - if there is a P-certificate for n,m in V, $n \leq h$
      - $O = O \cup$ <PRE-PREPARE,v+1,n,m>
    - Otherwise, if $n \leq h$ but no P-certificate:
      - $N = N \cup$ <PRE-PREPARE,v+1,n,null>
- p’ multicasts <NEW-VIEW,v+1,V,O,N>

View Change: Backup

- Backup accepts NEW-VIEW message for v+1 if
  - it is signed properly
  - it contains in V a valid VIEW-CHANGE message for v+1
  - it can verify locally that O is correct (repeating the primary's computation)
- Adds all entries in O to its log (so did p')
- Multicasts a PREPARE for each message in O
- Adds all PREPARE to log and enters new view

BFT Discussion

- Is PBFT practical?
- Does it address the concerns that enterprise users would like to be addressed?