PBFT: A Byzantine Renaissance

- Practical Byzantine Fault-Tolerance (CL99, CL00)
  - first to be safe in asynchronous systems
  - live under weak synchrony assumptions -Byzantine Paxos!
  - fast! PBFT uses MACs instead of public key cryptography
  - uses proactive recovery to tolerate more failures over system lifetime: now need no more than f failures in a "window"
- BASE (RCL 01)
  - uses abstraction to reduce correlated faults

The Setup

System Model
- Asynchronous system
- Unreliable channels

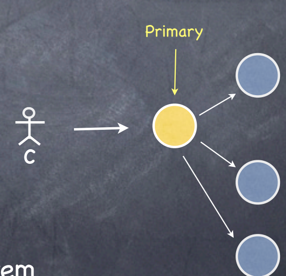
Crypto
- Public/Private key pairs
- MACs
- Collision-resistant hashes
- Unbreakable

Service
- Byzantine clients
- Up to f Byzantine servers
- $N > 3f$ total servers

System Goals
- Always safe
- Live during periods of synchrony

The General Idea

- Primary-backup + 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 (primary included)



What could possibly go wrong?

- The Primary could be faulty!
  - > could ignore commands; assign same sequence number to different requests; skip sequence numbers; etc
  - Backups monitor primary's behavior and trigger view changes to replace faulty primary
- Backups could be faulty!
  - > could incorrectly store commands forwarded by a correct primary
  - use dissemination Byzantine quorum systems [MR98]
- Faulty replicas could incorrectly respond to the client!

What could possibly go wrong?

- The Primary could be faulty!
  - could ignore commands; assign same sequence number to different requests; skip sequence numbers; etc
  - Backups monitor primary's behavior and trigger view changes to replace faulty primary
- Backups could be faulty!
  - could incorrectly store commands forwarded by a correct primary
  - use dissemination Byzantine quorum systems [MR98]
- Faulty replicas could incorrectly respond to the client!
  - Client waits for $f+1$ matching replies before accepting response

Me, or your lying eyes?

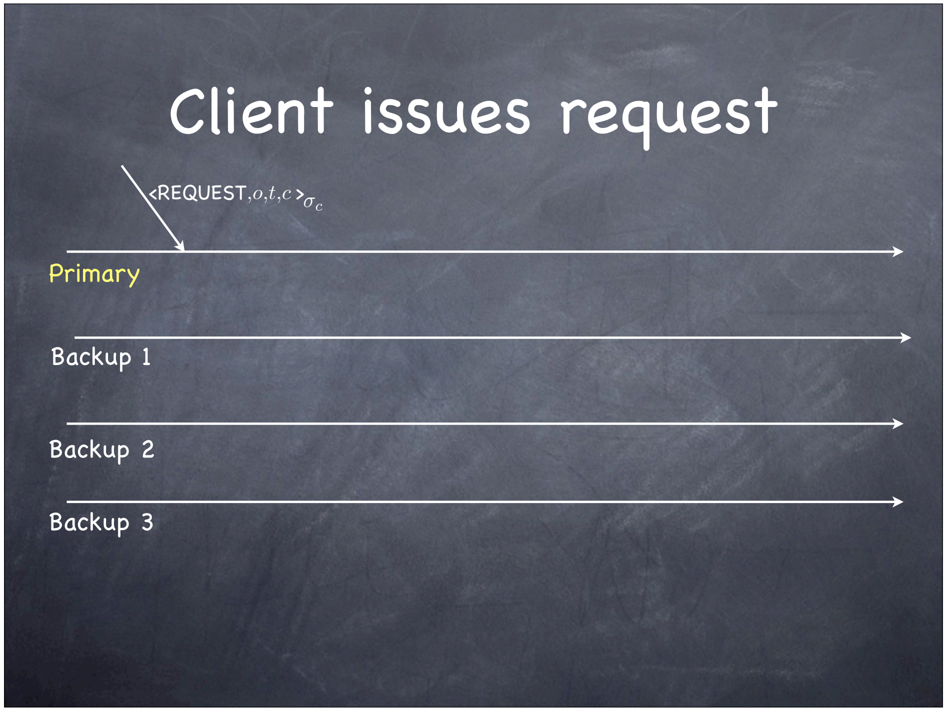
- Algorithm steps are justified by certificates
  - Sets (quorums) of signed messages from distinct replicas proving that a property of interest holds
- With quorums of size at least $2f+1$
  - Any two quorums intersect in at least one correct replica
  - Always one quorum contains only non-faulty replicas

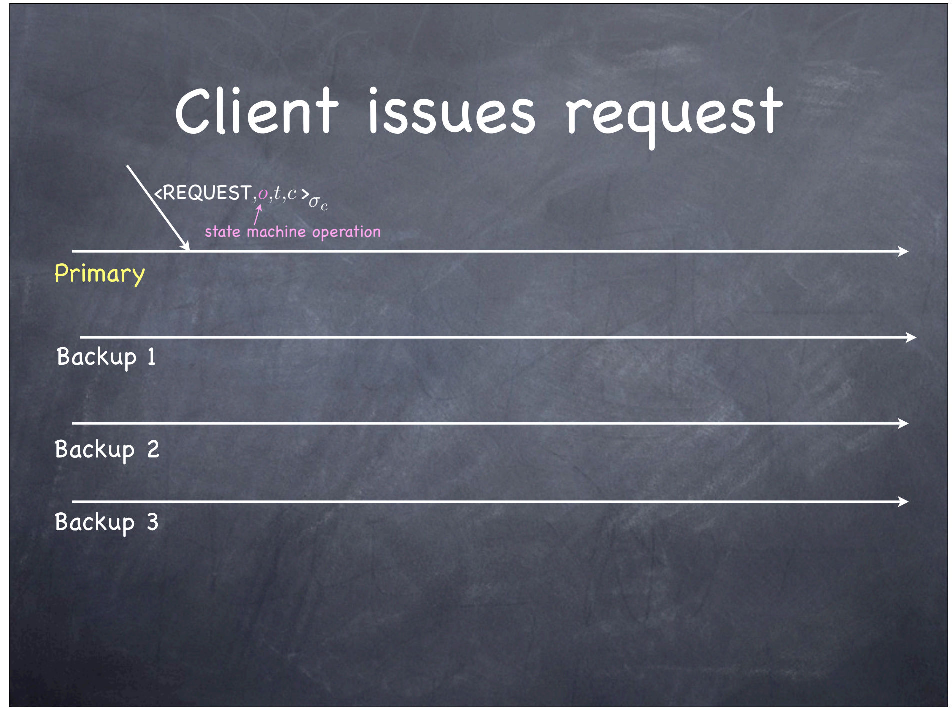
PBFT: The site map

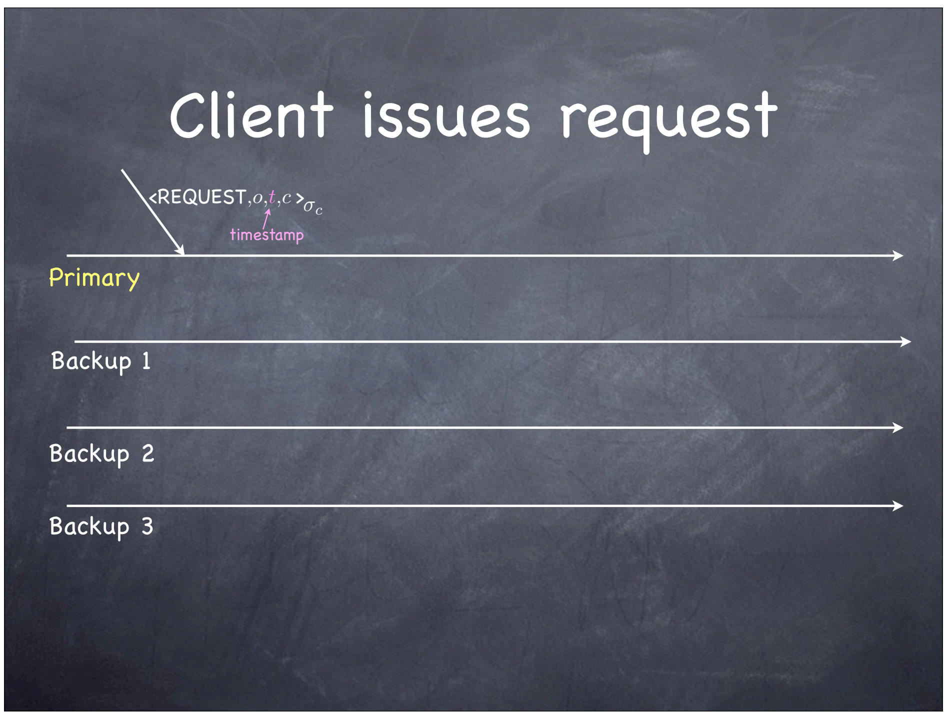
- Normal operation
  - How the protocol works in the absence of failures - hopefully, the common case
- View changes
  - How to depose a faulty primary and elect a new one
- Garbage collection
  - How to reclaim the storage used to keep certificates
- Recovery
  - How to make a faulty replica behave correctly again

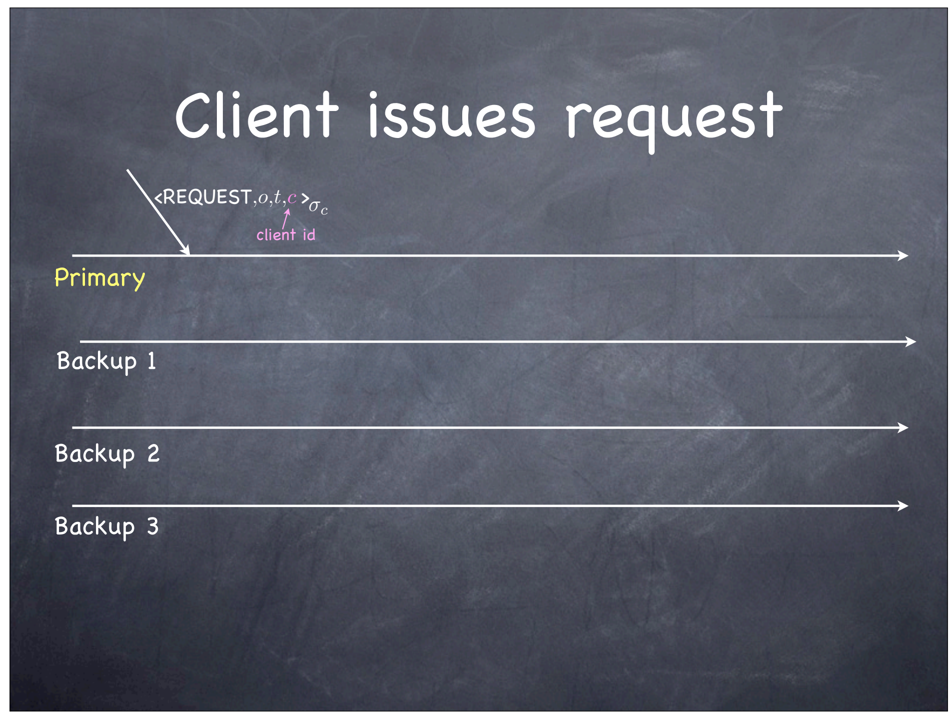
Normal Operation

- Three phases:
  - 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
- Each replica i maintains the following state:
  - Service state
  - A message log with all messages sent or received
  - An integer representing i's current view


Client issues request
$\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$
Primary
Backup 1
Backup 2
Backup 3

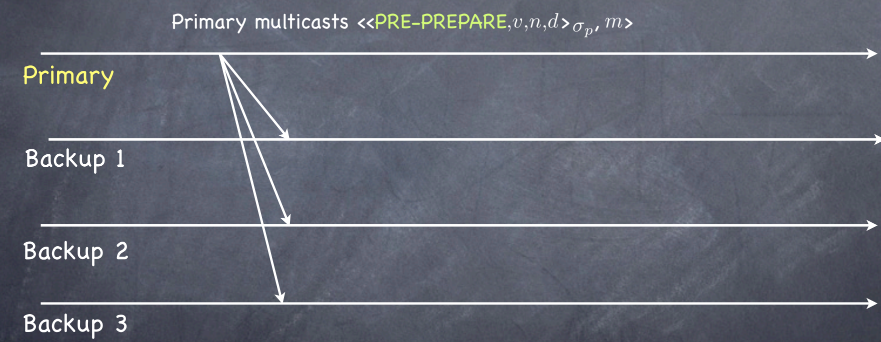

Client issues request
$\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$
state machine operation
Primary
Backup 1
Backup 2
Backup 3


Client issues request
$\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$
timestamp
Primary
Backup 1
Backup 2
Backup 3

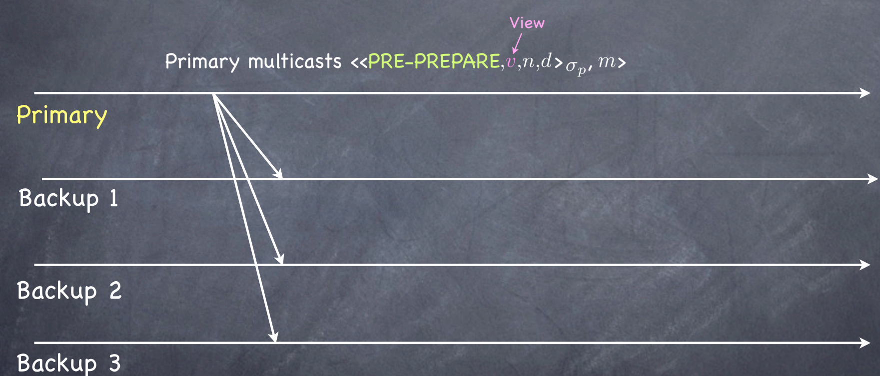

Client issues request
$\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$
client id
Primary
Backup 1
Backup 2
Backup 3

Client issues request

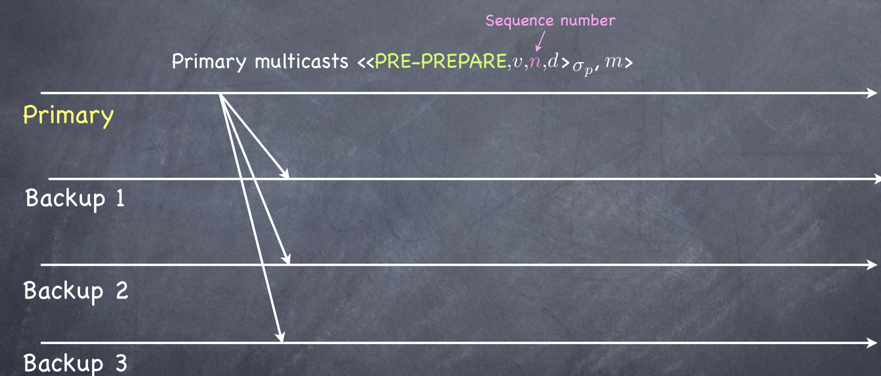
Pre-prepare

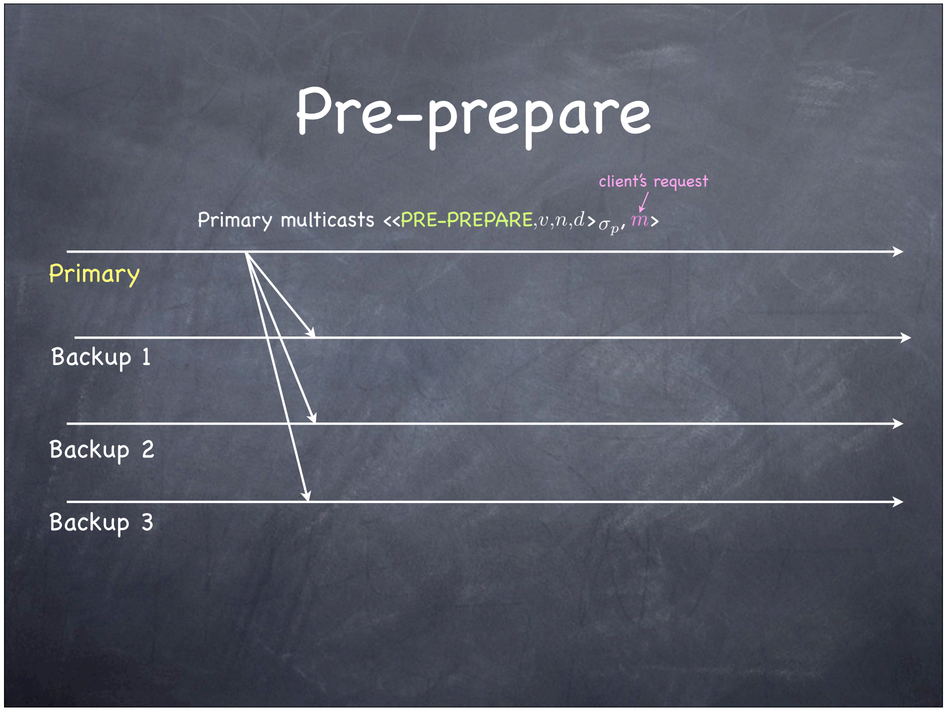


Pre-prepare

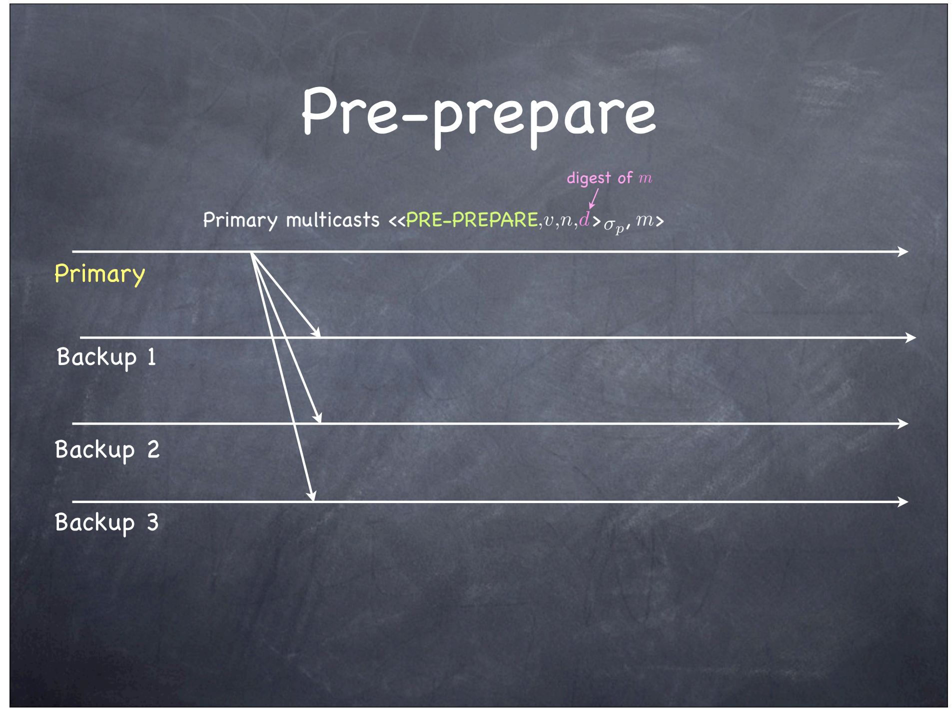


Pre-prepare

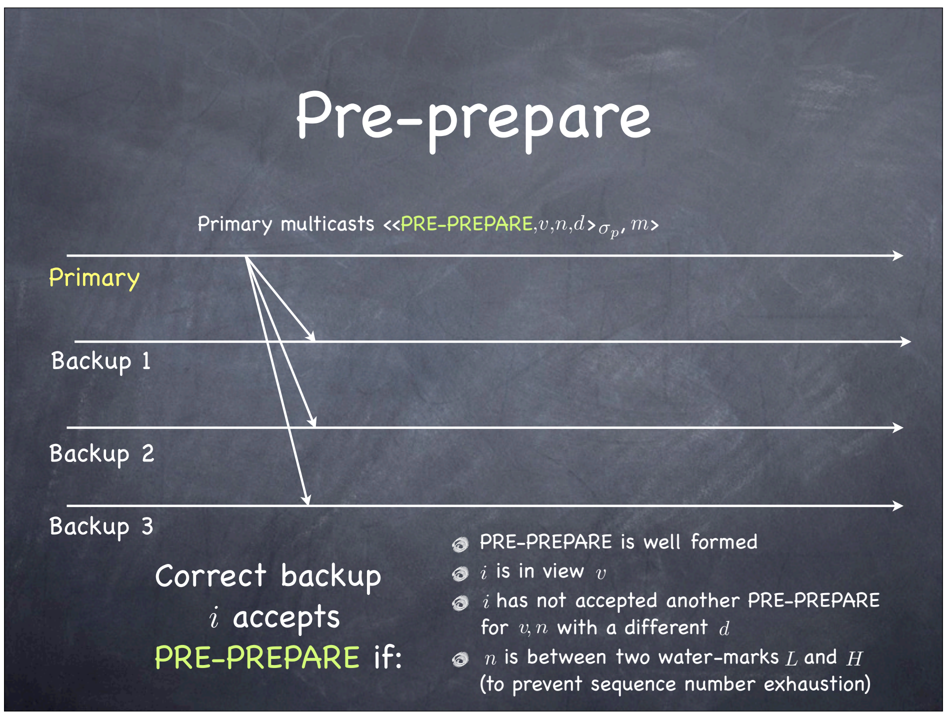




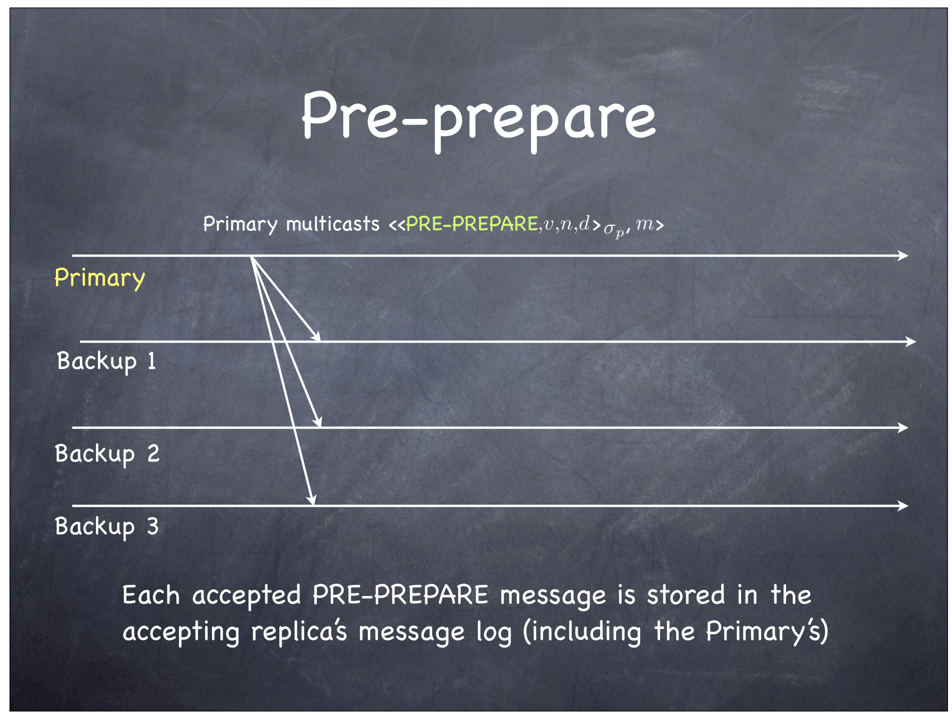
Pre-prepare
client's request
Primary multicasts $\langle\langle$PRE-PREPARE,v,n,$d\rangle_{\sigma_p}$, $m\rangle$
Primary
Backup 1
Backup 2
Backup 3



Pre-prepare
digest of m
Primary multicasts $\langle\langle$PRE-PREPARE,v,n,$d\rangle_{\sigma_p}$, $m\rangle$
Primary
Backup 1
Backup 2
Backup 3

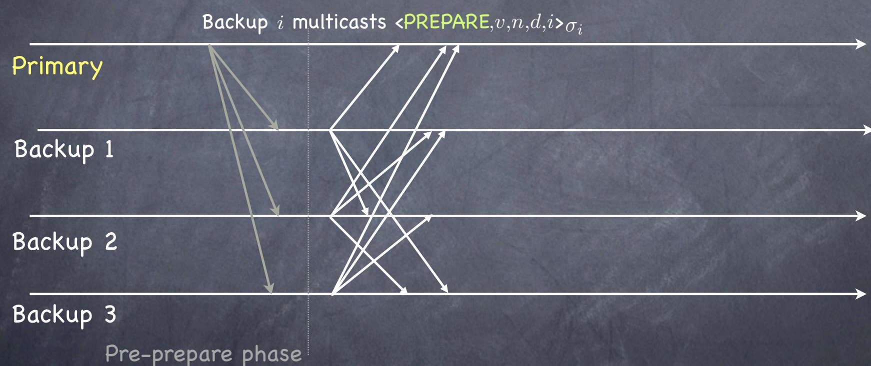


Pre-prepare
Primary multicasts $\langle\langle$PRE-PREPARE,v,n,$d\rangle_{\sigma_p}$, $m\rangle$
Primary
Backup 1
Backup 2
Backup 3
Correct backup i accepts PRE-PREPARE if:
PRE-PREPARE is well formed
i is in view v
i has not accepted another PRE-PREPARE for v, n with a different d
n is between two water-marks L and H (to prevent sequence number exhaustion)



Pre-prepare
Primary multicasts $\langle\langle$PRE-PREPARE,v,n,$d\rangle_{\sigma_p}$, $m\rangle$
Primary
Backup 1
Backup 2
Backup 3
Each accepted PRE-PREPARE message is stored in the accepting replica's message log (including the Primary's)

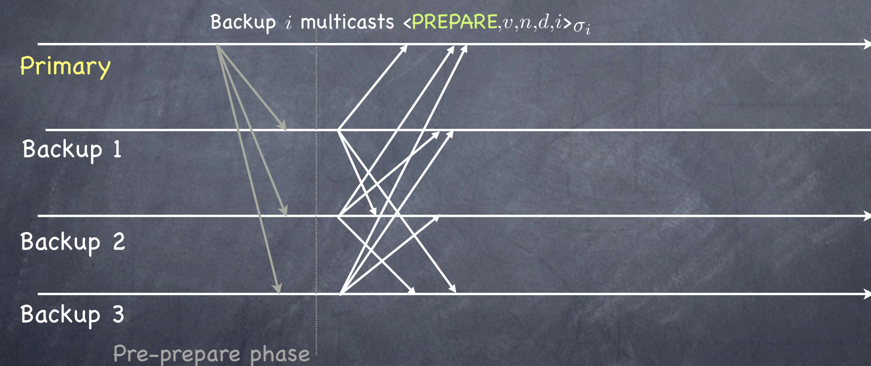
Prepare



Correct replica i accepts PREPARE if:

- PREPARE is well formed
- i is in view v
- n is between two water-marks L and H

Prepare



- Replicas that send PREPARE accept seq.# n for m in view v
- Each accepted PREPARE message is stored in the accepting replica's message log

Prepare Certificate

- P-certificates ensure total order within views

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
  - $2f$ PREPARE from different backups that match the pre-prepare

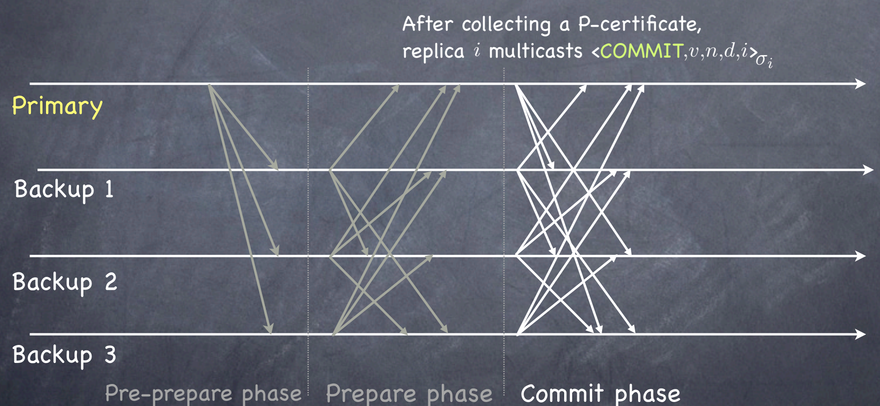
Prepare Certificate

- 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
  - $2f$ PREPARE 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(m_1,v,n) and P-certificate(m_2,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

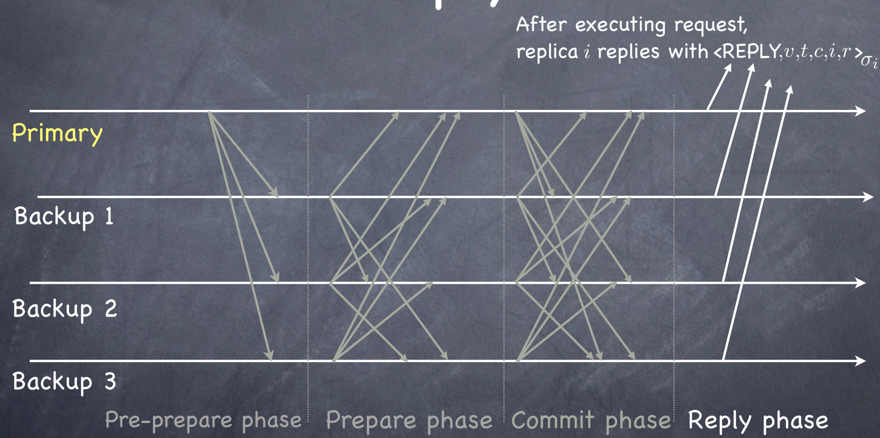
Commit



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 C-certificate for it, and has cleared all requests with smaller sequence numbers

Reply



Aux armes les backups!

- A disgruntled backup mutinies:
  - stops accepting messages (but for VIEW-CHANGE & NEW-VIEW)
  - multicasts <VIEW-CHANGE,$v+1, \mathcal{P}$ >$_{\sigma_i}$
  - $\mathcal{P}$ 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 $\mathcal{V}$, indicating support from $2f+1$ distinct replicas (including itself)

On to view $v+1$: the new primary

- The "primary elect" $\hat{p}$ (replica $v+1 \bmod N$) extracts from the new-view certificate $\mathcal{V}$:
  - the highest sequence number h of any message for which $\mathcal{V}$ contains a P-certificate

On to view $v+1$: the new primary

- The "primary elect" $\hat{p}$ (replica $v+1 \bmod N$) extracts from the new-view certificate $\mathcal{V}$:
  - the highest sequence number h of any message for which $\mathcal{V}$ contains a P-certificate

On to view $v+1$: the new primary

- The "primary elect" $\hat{p}$ (replica $v+1 \bmod N$) extracts from the new-view certificate $\mathcal{V}$:
  - the highest sequence number h of any message for which $\mathcal{V}$ contains a P-certificate
  - two sets $\mathcal{O}$ and $\mathcal{N}$:
    - If there is a P-certificate for n,m in $\mathcal{V}$, $n \leq h$
      $\mathcal{O} = \mathcal{O} \cup$ <PRE-PREPARE,$v+1,n,m$>$_{\sigma_{\hat{p}}}$
    - Otherwise, if $n \leq h$ but no P-certificate:
      $\mathcal{N} = \mathcal{N} \cup$ <PRE-PREPARE,$v+1,n,null$>$_{\sigma_{\hat{p}}}$

On to view $v+1$: the new primary

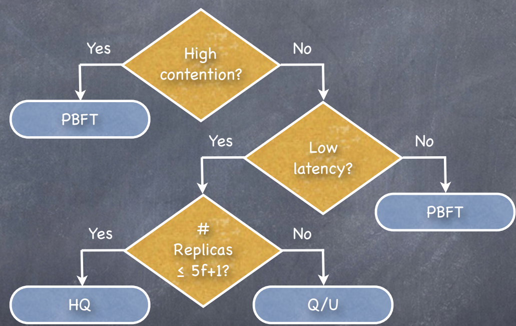
- The "primary elect" $\hat{p}$ (replica $v+1 \bmod N$) extracts from the new-view certificate $\mathcal{V}$:
  - the highest sequence number h of any message for which $\mathcal{V}$ contains a P-certificate
  - two sets $\mathcal{O}$ and $\mathcal{N}$:
    - If there is a P-certificate for n,m in $\mathcal{V}$,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

On to view $v+1$: the backup

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

Zyzzyva

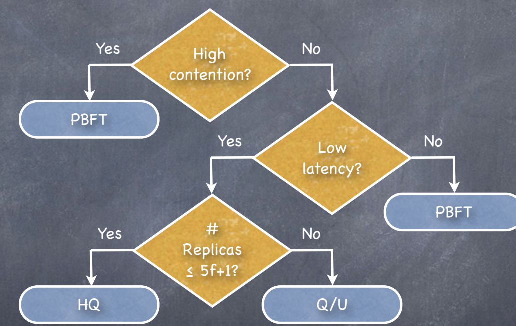
Why then another BFT protocol?



- Complex decision tree hampers BFT adoption

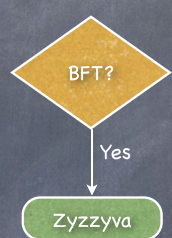
"Simplify, ~~simplify~~"

H.D. Thoreau



"Simplify, ~~simplify~~"

H.D. Thoreau



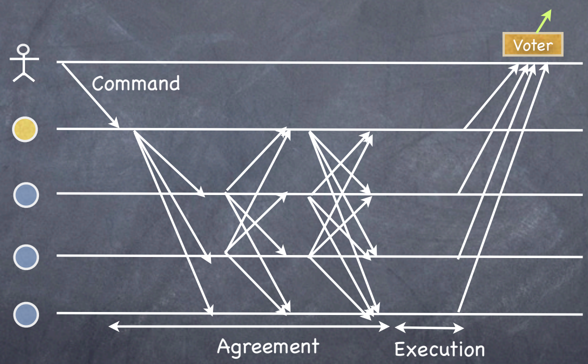
- <u>One</u> protocol that matches or tops its competitors in

  ✓ latency ✓ throughput ✓ cost of replication

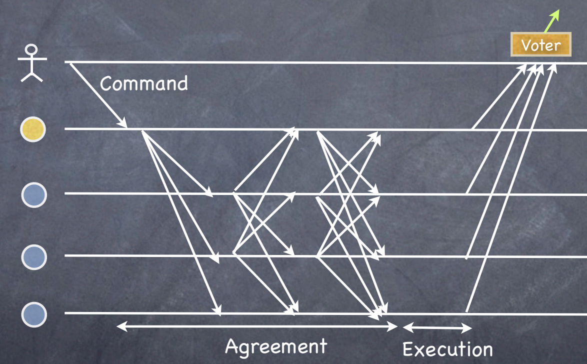
Replica coordination

- All correct replicas execute the same sequence of commands
- For each received command c, correct replicas:
  - Agree on c's position in the sequence
  - Execute c in the agreed upon order
  - Replies to the client

How it is done now



How Zyzzyva does it



Stability

- A command is stable at a replica once its position in the sequence cannot change

RSM Safety

Correct clients only process replies to stable commands

RSM Liveness

All commands issued by correct clients eventually become stable and elicit a reply

Enforcing safety

- RSM safety requires:
  - Correct clients only process replies to stable commands
- ...but RSM implementations enforce instead:
  - Correct replicas only execute and reply to commands that are stable
- Service performs an output commit with each reply

Speculative BFT: "Trust, but Verify"

- Insight: output commit at the client, not at the service!
- Replicas execute and reply to a command without knowing whether it is stable
  - trust order provided by primary
  - no explicit replica agreement!
- Correct client, before processing reply, verifies that it corresponds to stable command
  - if not, client takes action to ensure liveness

Verifying stability

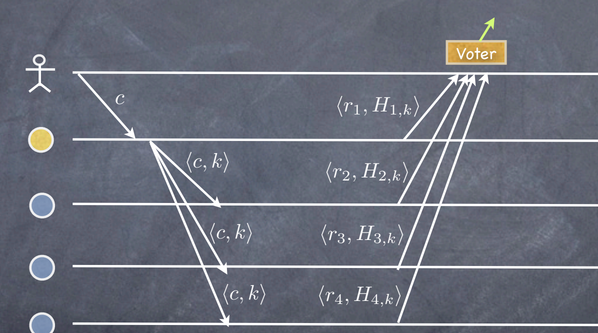
- Necessary condition for stability in Zyzzyva:

  A command c can become stable only if a majority of correct replicas agree on its position in the sequence
- Client can process a response for c iff:
  - a majority of correct replicas agrees on c's position
  - the set of replies is incompatible, for all possible future executions, with a majority of correct replicas agreeing on a different command holding c's current position

Command History

- $H_{i,k}$ = a hash of the sequence of the first k commands executed by replica i
- On receipt of a command c from the primary, replica appends c to its command history
- Replica reply for c includes:
  - the application-level response
  - the corresponding command history

Case 1: Unanimity



- Client processes response if all replies match:

  $r_1 = \ldots = r_4 \wedge H_{1,k} = \ldots = H_{4,k}$

Safe?

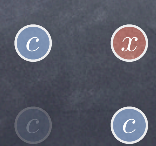
- ✓ A majority of correct replicas agrees on c's position (all do!)
- If primary fails
  - New primary determines k-th command by asking $n - f$ replicas for their H

Safe?

- ✓ A majority of correct replicas agrees on c's position (all do!)
- If primary fails
  - New primary determines k-th command by asking $n - f$ replicas for their H

Safe?

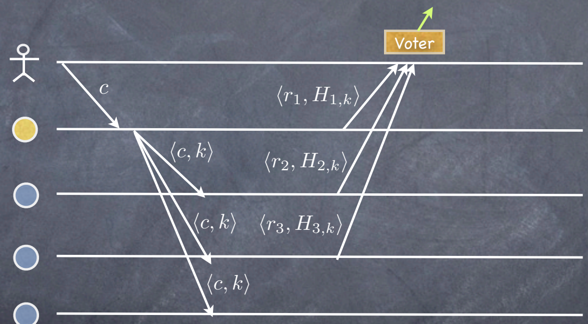
- ✓ A majority of correct replicas agrees on c's position (all do!)
- If primary fails
  - New primary determines k-th command by asking $n - f$ replicas for their H



Safe?

- ✓ A majority of correct replicas agrees on c's position (all do!)
- If primary fails
  - New primary determines c's position by asking $n - f$ replicas for their H
- ✓ It is impossible for a majority of correct replicas to agree on a different command for c's position

Case 2: A majority of correct replicas agree



- At least $2f+1$ replies match

Safe?

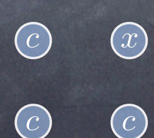
✓ A majority of correct replicas agrees on c's position

- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H

Safe?

✓ A majority of correct replicas agrees on c's position

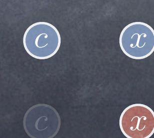
- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H



Safe?

✓ A majority of correct replicas agrees on c's position

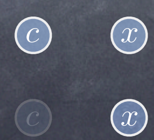
- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H



Safe?

✓ A majority of correct replicas agrees on c's position

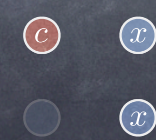
- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H



Safe?

✓ A majority of correct replicas agrees on c's position

- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H



Safe?

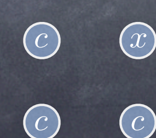
✓ A majority of correct replicas agrees on c's position

- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H

Safe?

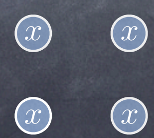
✓ A majority of correct replicas agrees on c's position

- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H



Safe?

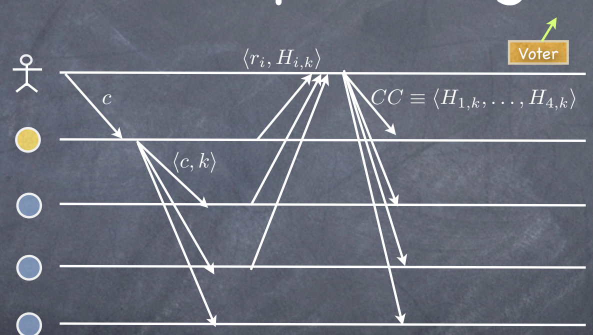
- ✓ A majority of correct replicas agrees on c's position
- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H



Safe?

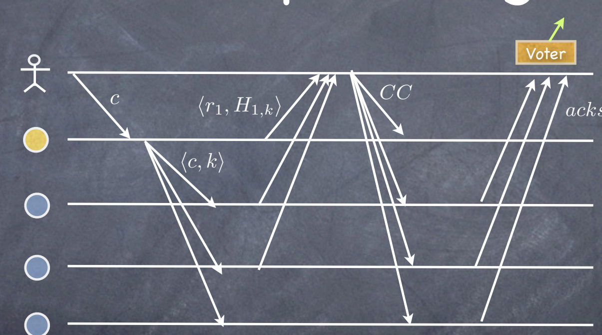
- ✓ A majority of correct replicas agrees on c's position
- If primary fails
  - New primary determines k-th command by asking $n-f$ replicas for their H
- Not safe!

Case 2: A majority of correct replicas agree



- Client sends to all a commit certificate containing $2f+1$ matching histories

Case 2: A majority of correct replicas agree



- Client processes response if it receives at least $2f+1$ acks

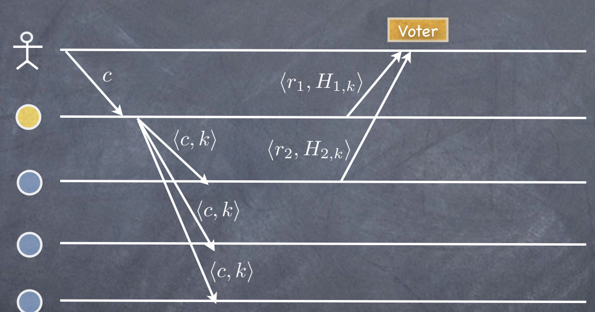
Safe?

- Certificate proves that a majority of correct replicas agreed on c's position
- If primary fails
  - New primary determines k-th command by contacting $n-f$ replicas
  - This set contains at least one correct replica with a copy of the certificate
- ✓ Incompatible with a majority backing a different command for that position

Stability and command histories

- Stability depends on matching command histories
- Stability is prefix-closed:
  - If a command with sequence number n is stable, then so is every command with sequence number $n' < n$

Case 3: None of the above

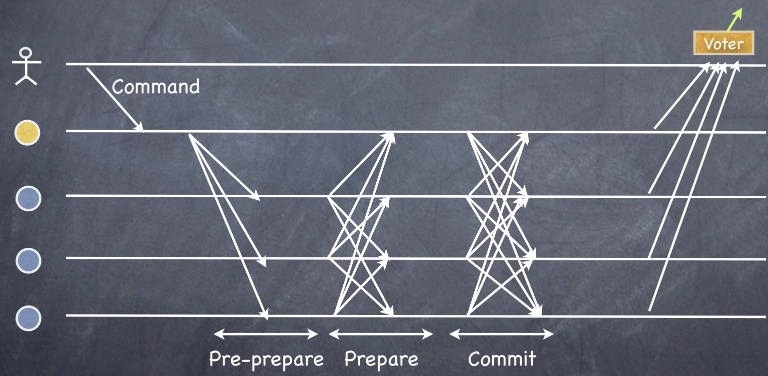


- Fewer than $2f+1$ replies match
- Clients retransmits c to all replicas-hinting primary may be faulty

Zyzzyva recap

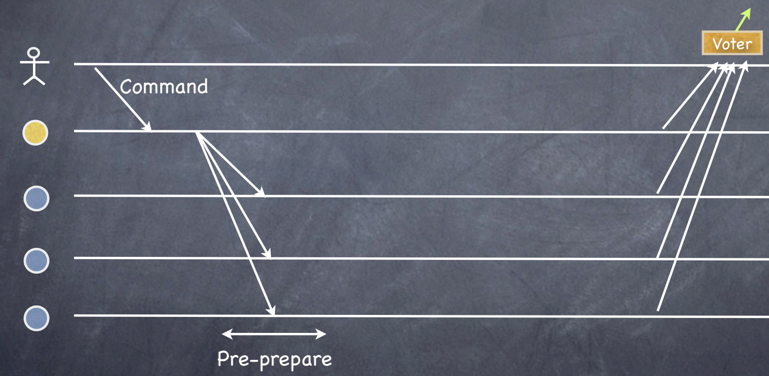
- Output commit at the client, not the service
- Replicas execute requests without explicit agreement
- Client verifies if response corresponds to stable command
- At most 2 phases within a view to make command stable

The Case of the Missing Phase



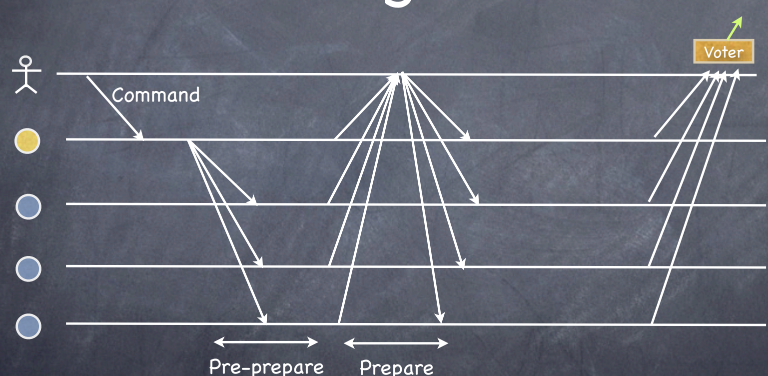
- Client processes response if it receives at least $f+1$ matching replies after commit phase

The Case of the Missing Phase



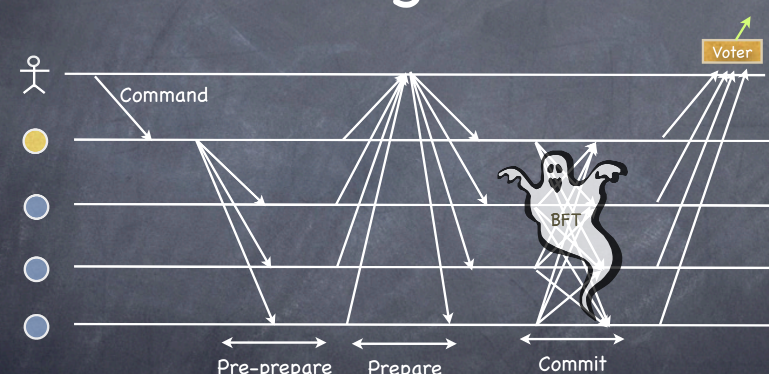
Unanimity

The Case of the Missing Phase



Majority

The Case of the Missing Phase



- Where did the third phase go?
- Why was it there to begin with?

View-Change: replacing the primary

- In PBFT, a replica that suspects primary is faulty goes unilaterally on strike
  - Stops processing messages in the view
  - Third "Commit" phase needed for liveness

View-Change: replacing the primary

- In PBFT, a replica that suspects primary is faulty goes unilaterally on strike
  - Stops processing messages in the view
  - Third "Commit" phase needed for liveness

- In Zyzzyva, the replica goes on "Technion strike"
  - Broadcasts "I hate the primary" and keeps on working
  - Stops when sees enough hate mail to ensure all correct replica will stop as well
- Extra phase is moved to the uncommon case

Faulty clients can't affect safety

- Faulty clients cannot create inconsistent commit certificates
  - Clients cannot fabricate command histories, as they are signed by replicas
  - It is impossible to generate a valid commit certificate that conflicts with the order of any stable request
    - Stability is prefix closed!

"Olly Olly Oxen Free!"

or, faulty clients can't affect liveness

"Olly Olly Oxen Free!"

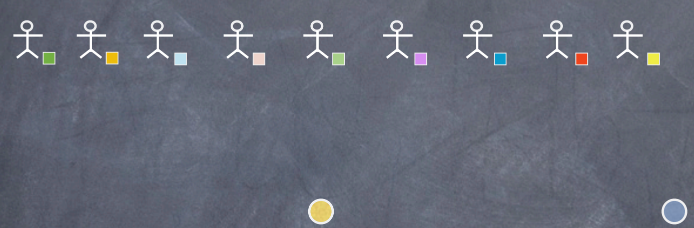
or, faulty clients can't affect liveness

- Faulty client omits to send CC for c
- Replicas commit histories are unaffected!
- Later correct client who establishes $c' > c$ is stable "frees" c as well
  - Stability is prefix closed!

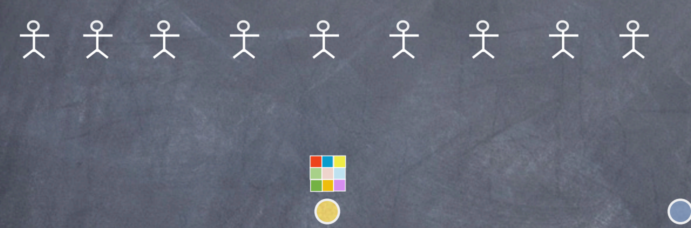
Optimizations

- Checkpoint protocol to garbage collect histories
- Optimizations include:
  - Replacing digital signatures with MAC
  - Replicating application state at only $2f+1$ replicas
  - Batching
  - Zyzzyva5

Batching



Batching



- Only one history digest for all requests in the batch-amortizes crypto operations

Throughput

Throughput

	Best case
PBFT	62K
QU	24K
HQ	15K
Zyzzyva	80K

BFT: From Z To A

Zyzzyva

BFT: From Z To A

Aardvark

Making Byzantine
Fault Tolerant Systems
Tolerate Byzantine Faults

Paved with good intentions

- No BFT protocol should rely on synchrony for safety
- FLP: No consensus protocol can be both safe and live in an asynchronous system
  - All one can guarantee is eventual progress

Paved with good intentions

- No BFT protocol should rely on synchrony for safety
- FLP: No consensus protocol can be both safe and live in an asynchronous system
  - All one can guarantee is eventual progress
- "Handle normal and worst case separately as a rule, because the requirements for the two are quite different:
  the normal case must be fast;
  the worst case must make some progress"
  -- Butler Lampson, "Hints for Computer System Design"

The road more traveled

- Maximize performance when
  - the network is synchronous
  - all clients and servers behave correctly
- While remaining
  - safe if at most f servers fail
  - eventually live

The Byzantine Empire (565 AD)

Synchronous, no failures

Asynchronous

Synchronous, with faults!

The Byzantine Empire (circa 2009 AD)

Synchronous, with or without failures

Asynchronous

Recasting the problem

- Misguided
- Maximize performance when
  - the network is synchronous
- Dangerous
  - all clients and servers behave correctly
- While remaining
- Futile
  - safe if at most f servers fail
  - eventually live

Recasting the problem

- Misguided
  - it encourages systems that fail to deliver BFT
- Dangerous
- Futile

Recasting the problem

- Misguided
  - it encourages systems that fail to deliver BFT
- Dangerous
  - it encourages fragile optimizations
- Futile

Recasting the problem

- Misguided
  - it encourages systems that fail to deliver BFT
- Dangerous
  - it encourages fragile optimizations
- Futile
  - it yields diminishing return on common case

BFT: a blueprint

- Build the system around execution path that:
  - provides acceptable performance across the broadest set of executions
  - it is easy to implement
  - it is robust against Byzantine attempts to push the system away from it