Byzantine Fault Tolerance

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 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 commits to a quorum

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

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 to foil the protocol?

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 Kaiyuan "the grade file is ready"

- Kaiyuan:

  cat grade

Design 1

- client, n servers
- client sends request to all of them
- waits for all n to reply
- only proceeds if all n agree

- what is wrong with this design?

Design 2

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

- what is wrong with design 2?

Issues with Design 2

- 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

Design 3

- 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
  - clients notify replicas of each operation, as well as primary; if no progress, force change of primary
  - replicas exchange info about ops sent by primary
  - replicas send results directly to client

PBFT: Overview

- Normal operation: how the protocol works in the absence of failures
- 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

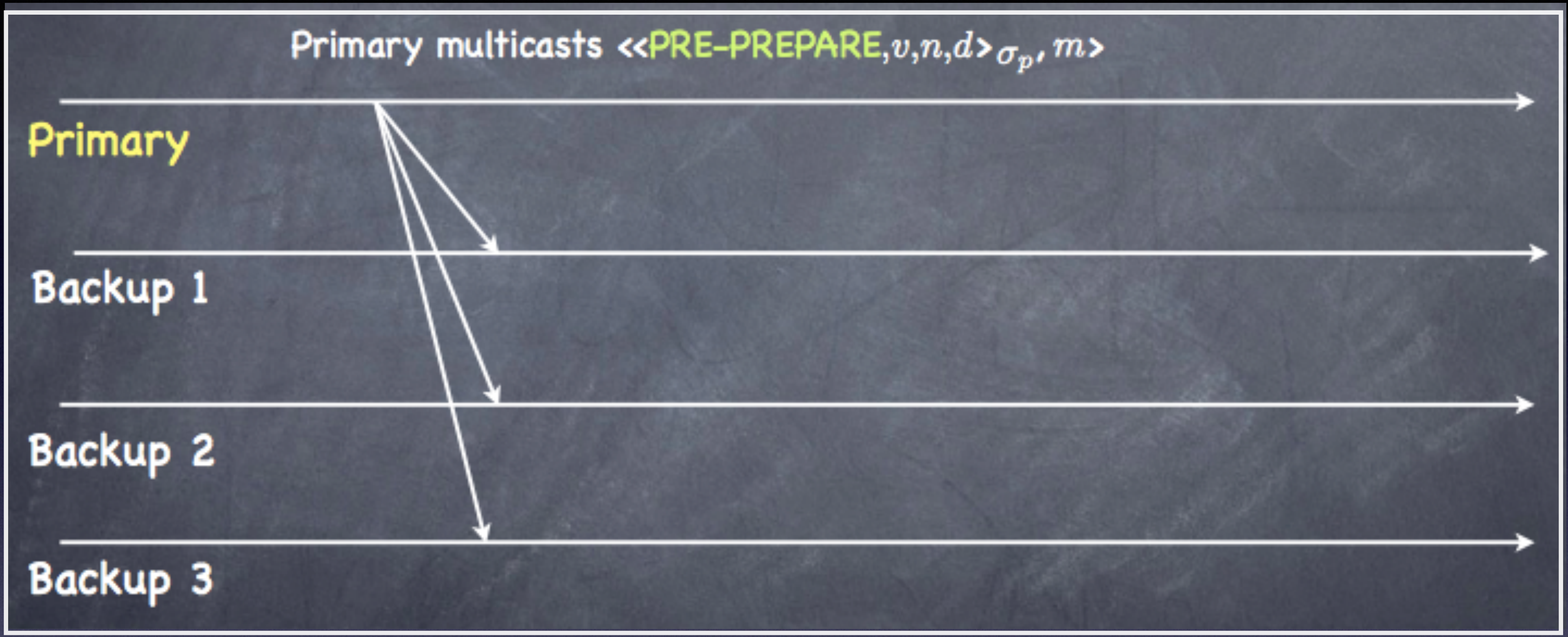
Normal Operation

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

Client issues request

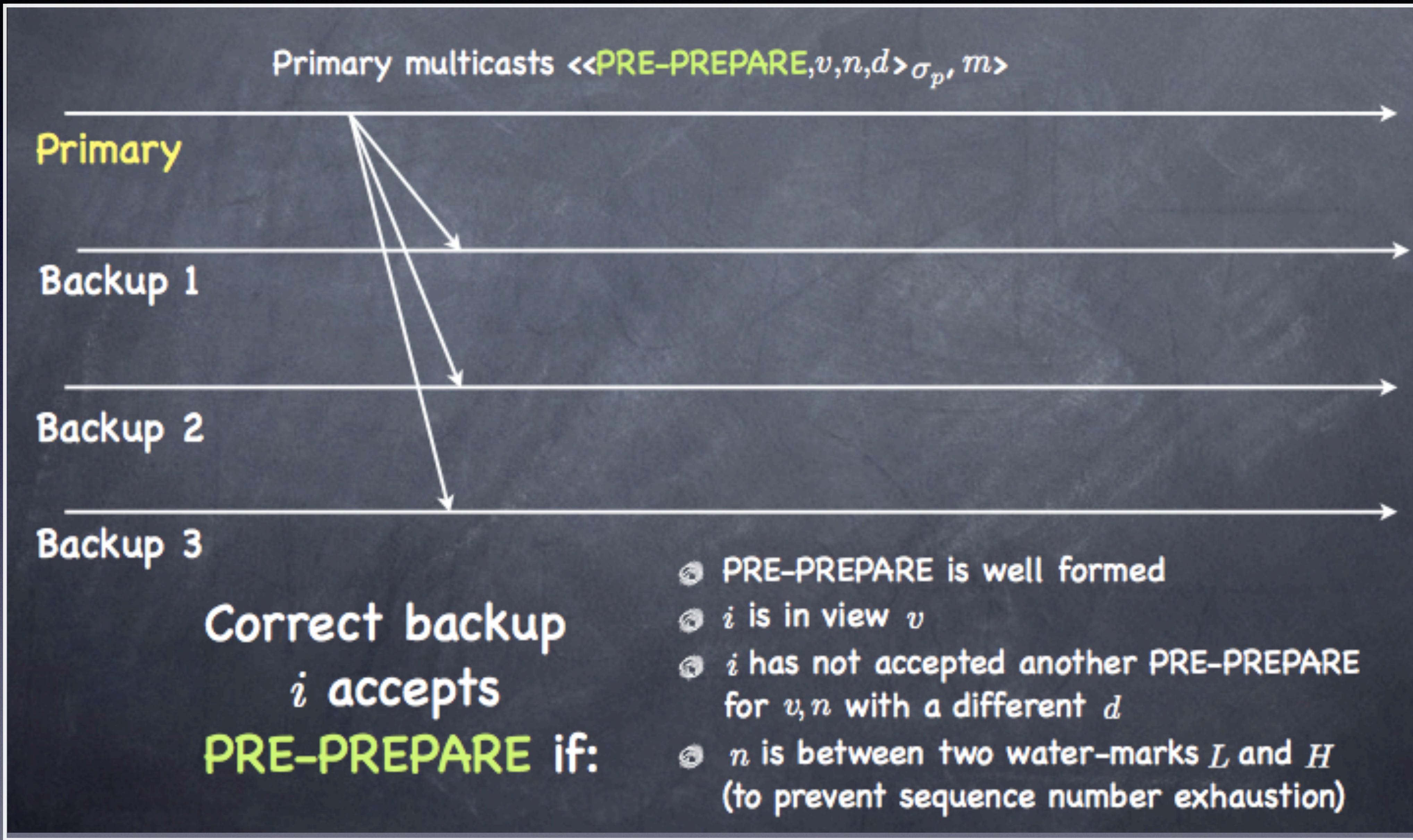
- o: state machine operation
- t: timestamp
- c: client id

Pre-prepare

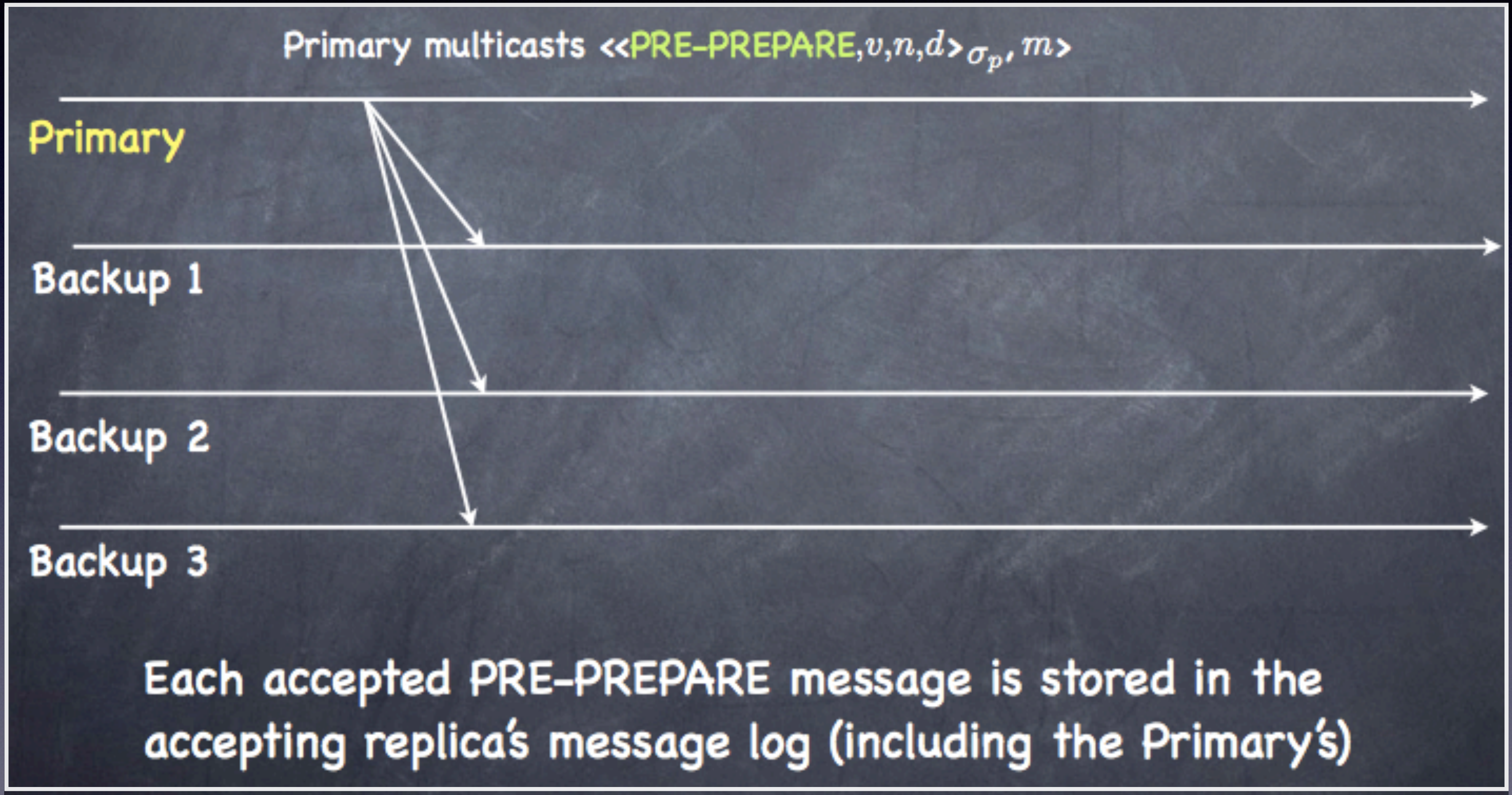


- v: view
- n: sequence number
- d: digest of m
- m: client's request

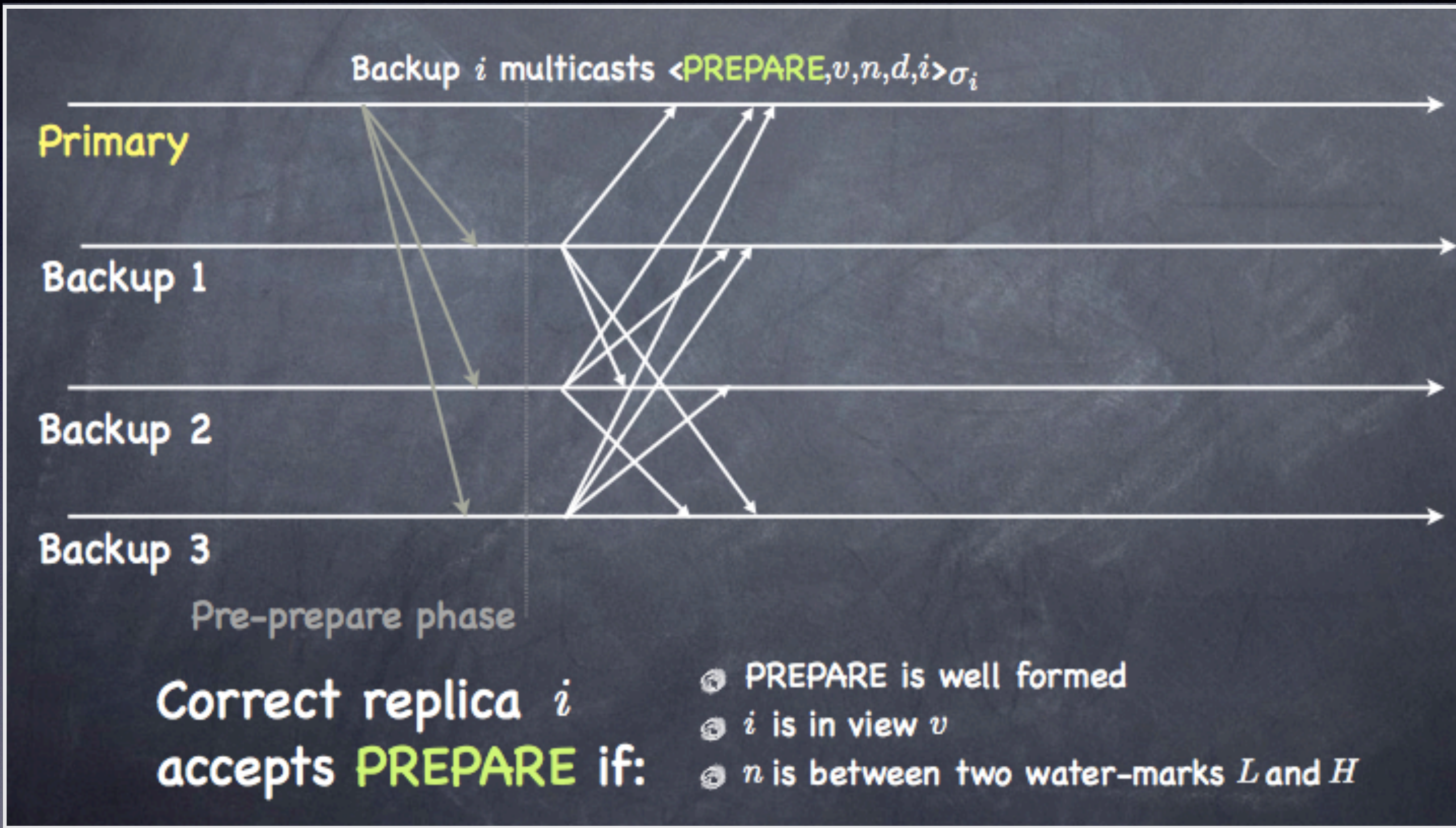
Pre-prepare Receipt



Pre-prepare Logging



Prepare



Backup i multicasts <PREPARE,v,n,d,i>$_{\sigma_i}$
Primary
Backup 1
Backup 2
Backup 3
Pre-prepare phase
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

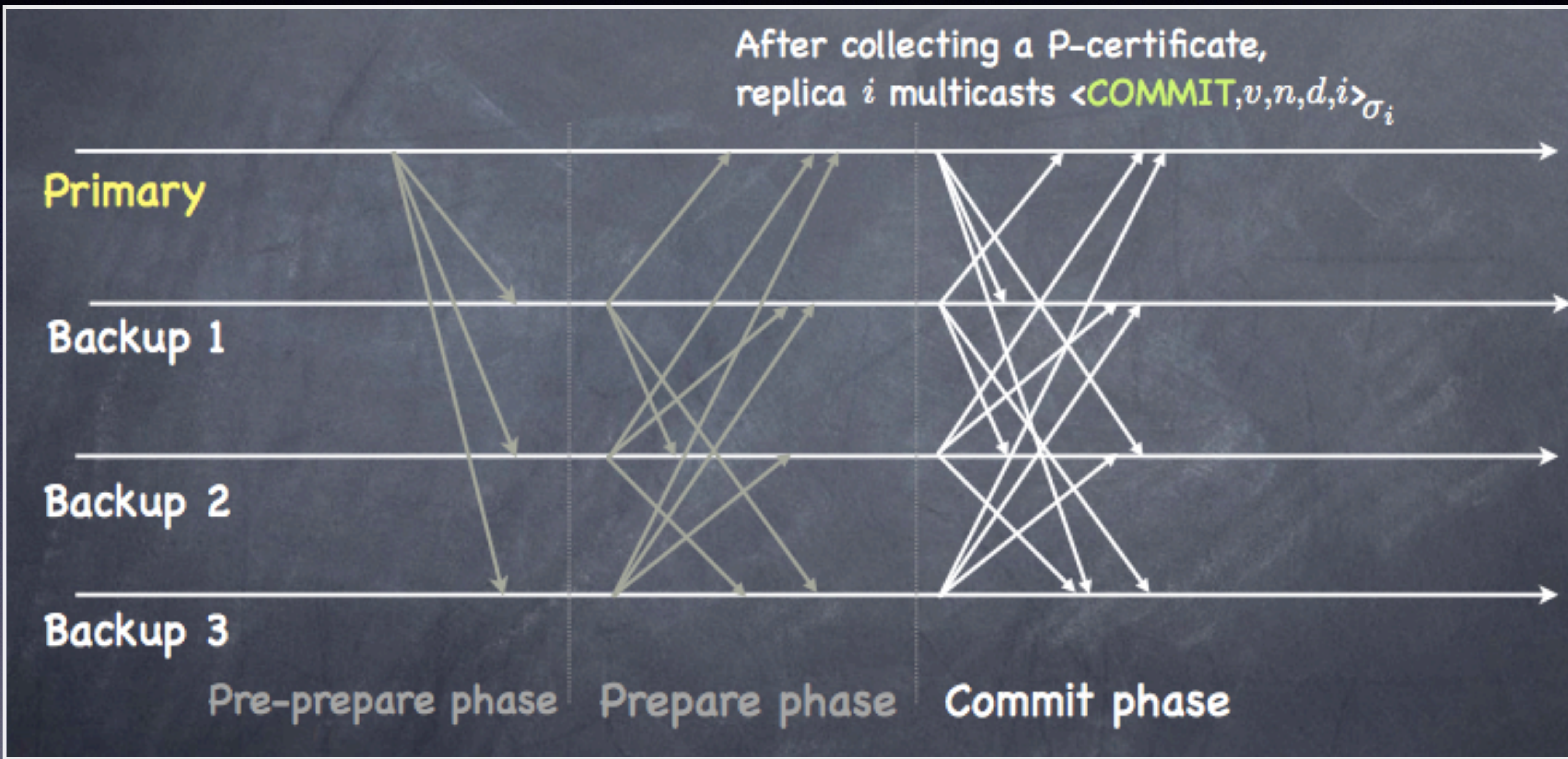
Prepare Certificate

- P-certificates ensure total order within views
- 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

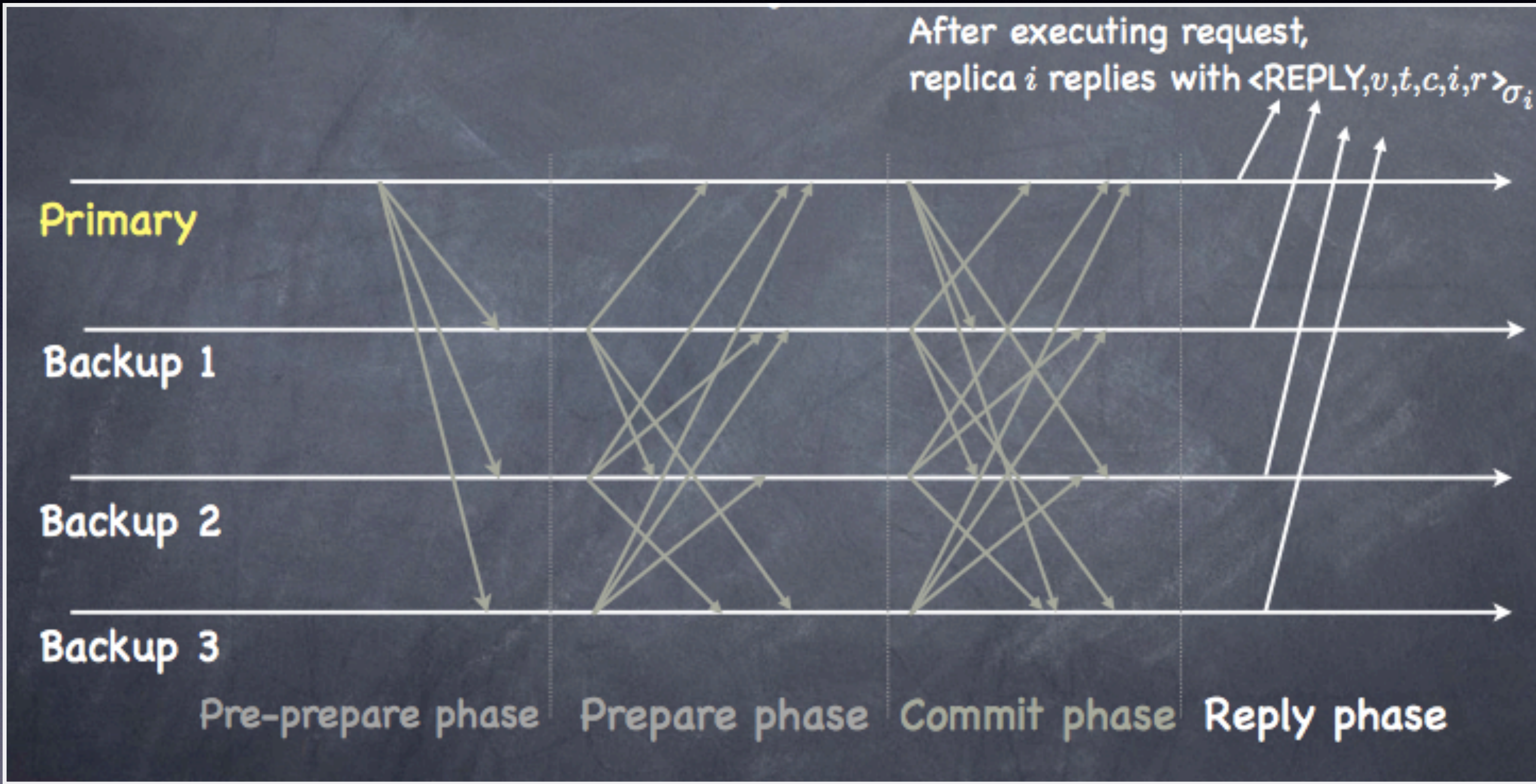


After collecting a P-certificate,
replica i multicasts <COMMIT,v,n,d,i>$_{\sigma_i}$
Primary
Backup 1
Backup 2
Backup 3
Pre-prepare phase
Prepare phase
Commit phase

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 <REPLY,v,t,c,i,r>$_{\sigma_i}$
Primary
Backup 1
Backup 2
Backup 3
Pre-prepare phase
Prepare phase
Commit phase
Reply phase

BFT Discussion

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

Bitcoin

- a digital currency
- a public ledger to prevent double-spending
- no centralized trust or mechanism <-- this is hard!

Why digital currency?

- might make online payments easier
- credit cards have worked well but aren't perfect
  - insecure -> fraud -> fees, restrictions, reversals
  - record of all your purchases

What is hard technically?

- forgery
- double spending
- theft

Idea

- Signed sequence of transactions
  - there are a bunch of coins, each owned by someone
  - every coin has a sequence of transaction records
    - one for each time this coin was transferred as payment
  - a coin's latest transaction indicates who owns it now

Transaction Record

- pub(user1): public key of new owner
- hash(prev): hash of this coin's previous transaction record
- sig(user2): signature over transaction by previous owner's private key

- BitCoin has more complexity: amount (fractional), multiple in/out, ...

Transaction Example

1. Y owns a coin, previously given to it by X:
   - T7: pub(Y), hash(T6), sig(X)
2. Y buys a hamburger from Z and pays with this coin
   - Z sends public key to Y
   - Y creates a new transaction and signs it
   - T8: pub(Z), hash(T7), sig(Y)
3. Y sends transaction record to Z
4. Z verifies: T8's sig() corresponds to T7's pub()
5. Z gives hamburger to Y

Double Spending

- Y creates two transactions for same coin: Y->Z, Y->Q
  - both with hash(T7)
- Y shows different transactions to Z and Q
- both transactions look good, including signatures and hash
- now both Z and Q will give hamburgers to Y

Defense

- publish log of all transactions to everyone, in same order
  - so Q knows about Y->Z, and will reject Y->Q
  - a "public ledger"
- ensure Y can't un-publish a transaction

Strawman Solution

- Assume a p2p network
- Peers flood new transactions over “overlay”
- Transaction is acceptable only if majority of peers think it is valid

- What are the issues with this scheme?

BitCoin Block Chain

- the block chain contains transactions on all coins
- many peers, each with a complete copy of the chain
  - proposed transactions flooded to all peers
  - new blocks flooded to all peers
- each block: hash(prevblock), set of transactions, *nonce*, current wall clock timestamp
- new block about ~10 minutes containing new xactions
- payee doesn't verify until xaction is in the block chain

“Mining” Blocks

- requirement: hash(block) has N leading zeros
  - each peer tries nonce values until this works out
  - trying one nonce is fast, but most nonces won't work
  - mining a block *not* a specific fixed amount of work
- one node can take months to create one block
  - but thousands of peers are working on it
  - such that expected time to first to find is about 10 minutes
- the winner floods the new block to all peers
- there is an incentive to mine a block — 12.5bc

Timing

- start: all peers know till B5
  - and are working on B6 (trying different nonces)
- Y sends Y->Z transaction to peers, which flood it
- peers buffer the transaction until B6 is computed
- peers that heard Y->Z include it in next block
- so eventually block chain is: B5, B6, B7, where B7 includes Y->Z

Double Spending

- what if Y sends out Y->Z and Y->Q at the same time?
  - no correct peer will accept both
  - a block will have one but not both
  - but there could be a fork: B6<-BZ and B6<-BQ

Forked Chain

- each peer believes whichever of BZ/BQ it saw first
- tries to create a successor
- if many more saw BZ than BQ, more will mine for BZ
  - so BZ successor likely to be created first
- even otherwise one will be extended first given significant variance in mining success time
- peers always switch to mining the longest fork, reinforcing agreement

Double Spending Defense

- wait for enough blocks to be minted
  - if a few blocks have been minted, unlikely that a different fork will win
  - if selling a high-value item, then wait for a few blocks before shipping
- could attacker start a fork from an old block?
  - yes -- but fork must be longer in order for peers to accept it
  - if the attacker has 1000s of CPUs -- more than all the honest bitcoin peers -- then the attacker can create the longest fork
  - system works only if no entity controls a majority of nodes

BitCoin Summary

- Key idea: block chain
- Public ledger is a great idea
- Decentralization might be good
- Mining is a clever way to avoid sybil attacks
- Question: Will BitCoin scale well?