Security
(and finale)

Dan Ports, CSEP 552

Today

- Security:
  what if parts of your distributed system are malicious?
  - BFT: state machine replication
  - Bitcoin: peer-to-peer currency
- Course wrap-up

Security

- Too broad a topic to cover here!
- Lots of security issues in distributed systems
- Focus on one today:
  how do we build a trusted distributed system when some of its components are untrusted?

Failure models

- Before: fail-stop
  nodes either execute the protocol correctly or just stop
- Now: Byzantine failures
  - some subset of nodes are faulty
  - they can behave *in any arbitrary way*:
    send messages, try to trick other nodes, collude, …
- Why this model?
  - if we can tolerate this, we can tolerate anything else:
    either malicious attacks or random failures

What can go wrong?

- Consider an unreplicated kv store:
- A: Append(x, "foo"); Append(x, "bar")
  B: Get(x) -> "foo bar"
  C: Get(x) -> "foo bar”
- What can a malicious server do?
  - return something totally unrelated
  - reorder the append operations (“bar foo”)
  - only process one of the appends
  - show B and C different results

What about Paxos?

- Paxos tolerates up to f out of 2f+1 *fail-stop* failures
- What could a malicious replica do?
  - stop processing requests (but Paxos should handle this!)
  - change the value of a key
  - acknowledge an operation then discard it
  - execute and log a different operation
  - tell some replicas that seq 42 is Put and others that it's Get
  - get different replicas into different views
  - force view changes to keep the system from making progress

BFT replication

- Same replicated state machine model as Paxos/VR
- assume 2f+1 out of 3f+1 replicas are non-faulty
- use voting, signatures to select the right results

BFT model

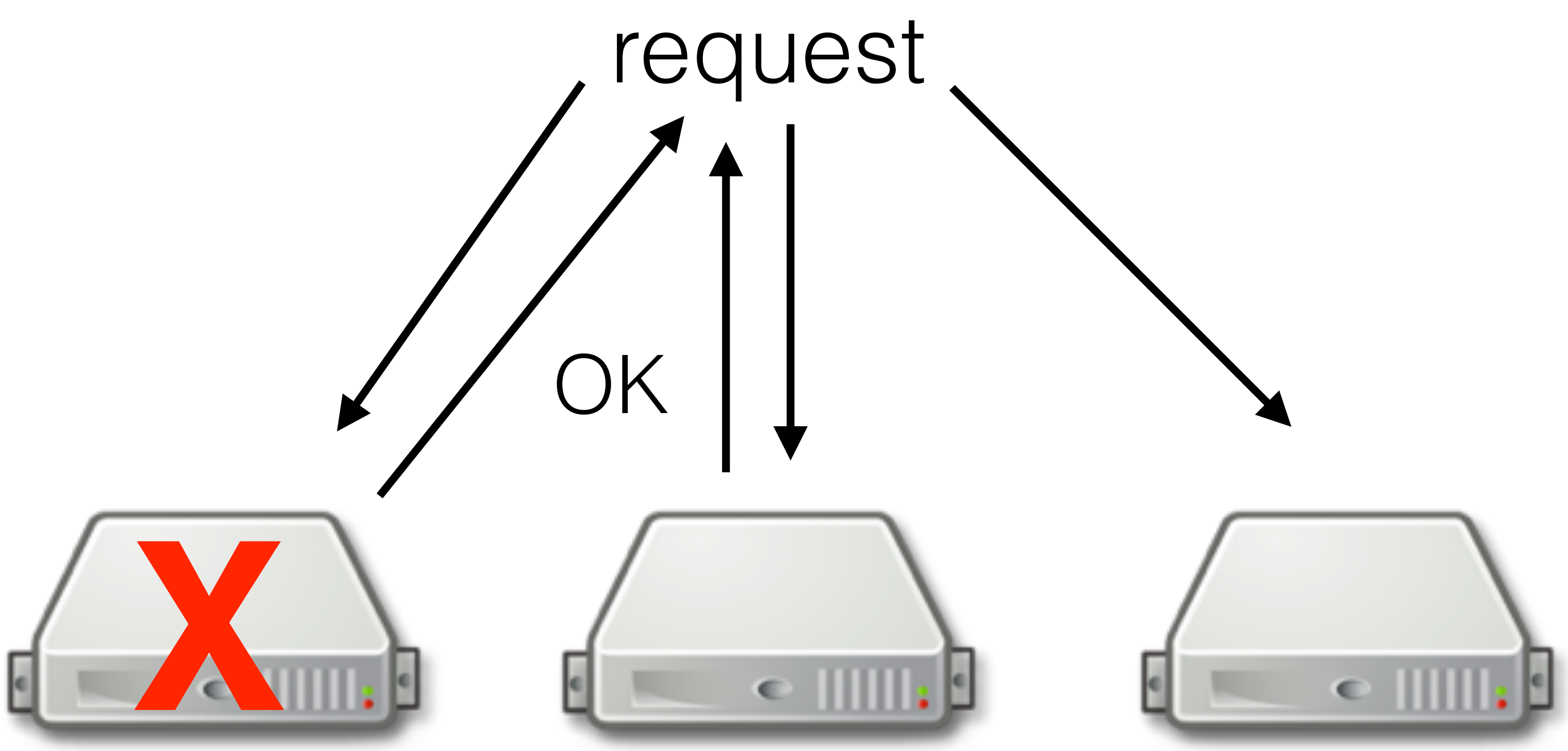
- attacker controls f replicas
  - can make them do anything
  - knows their crypto keys, can send messages
- attacker knows what protocol the other replicas are running
- attacker can delay messages in the network arbitrarily
- but the attacker can't
  - cause more than f replicas to fail
  - cause clients to misbehave break crypto

Why is BFT consensus hard?

- and why do we need 3f+1 replicas?

Paxos Quorums

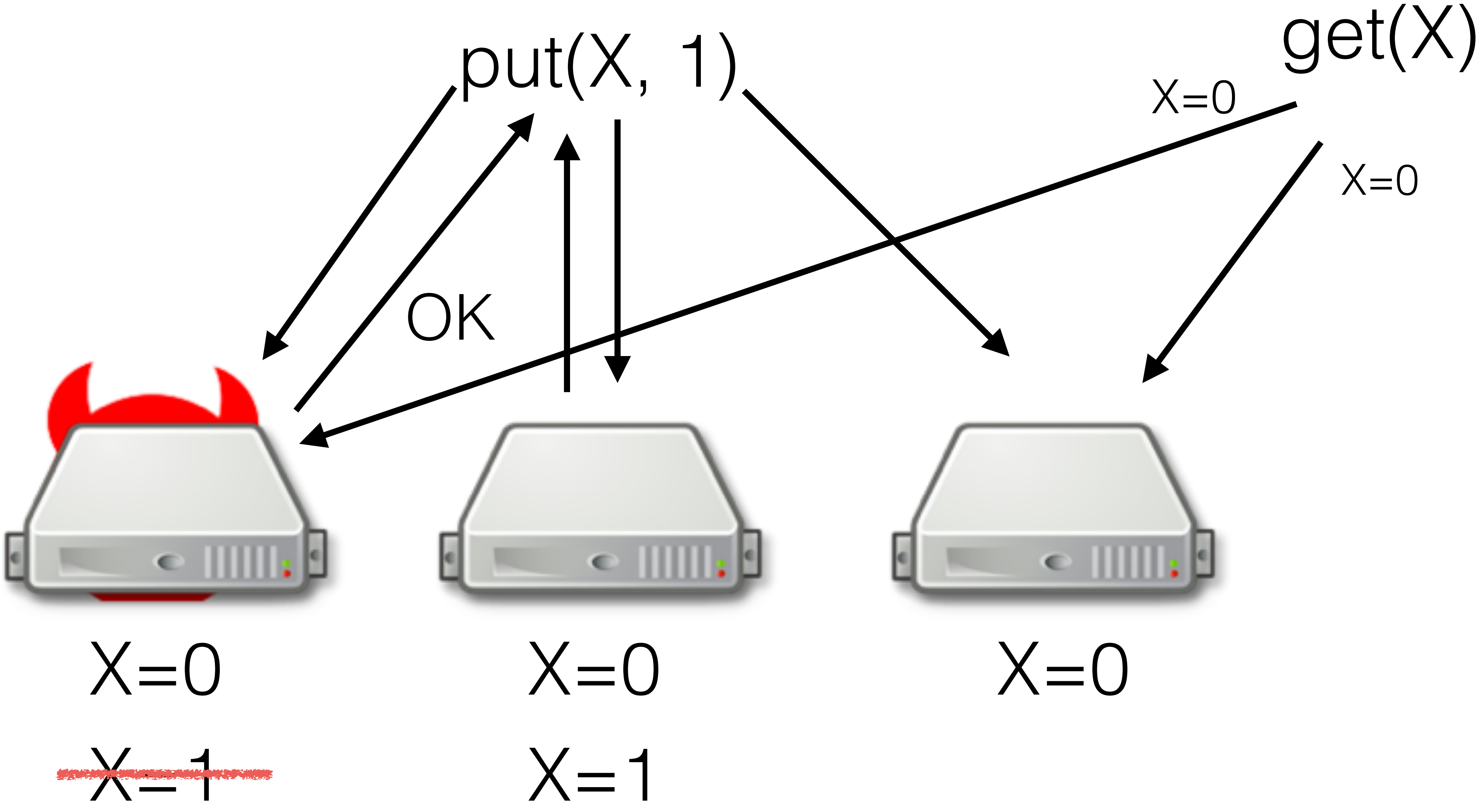
- Why did Paxos need 2f+1 replicas to tolerate f failures?
- Every operation needs to talk w/ a majority (f+1)



- f of those nodes might fail
- need one left
- quorums intersect

The Byzantine case

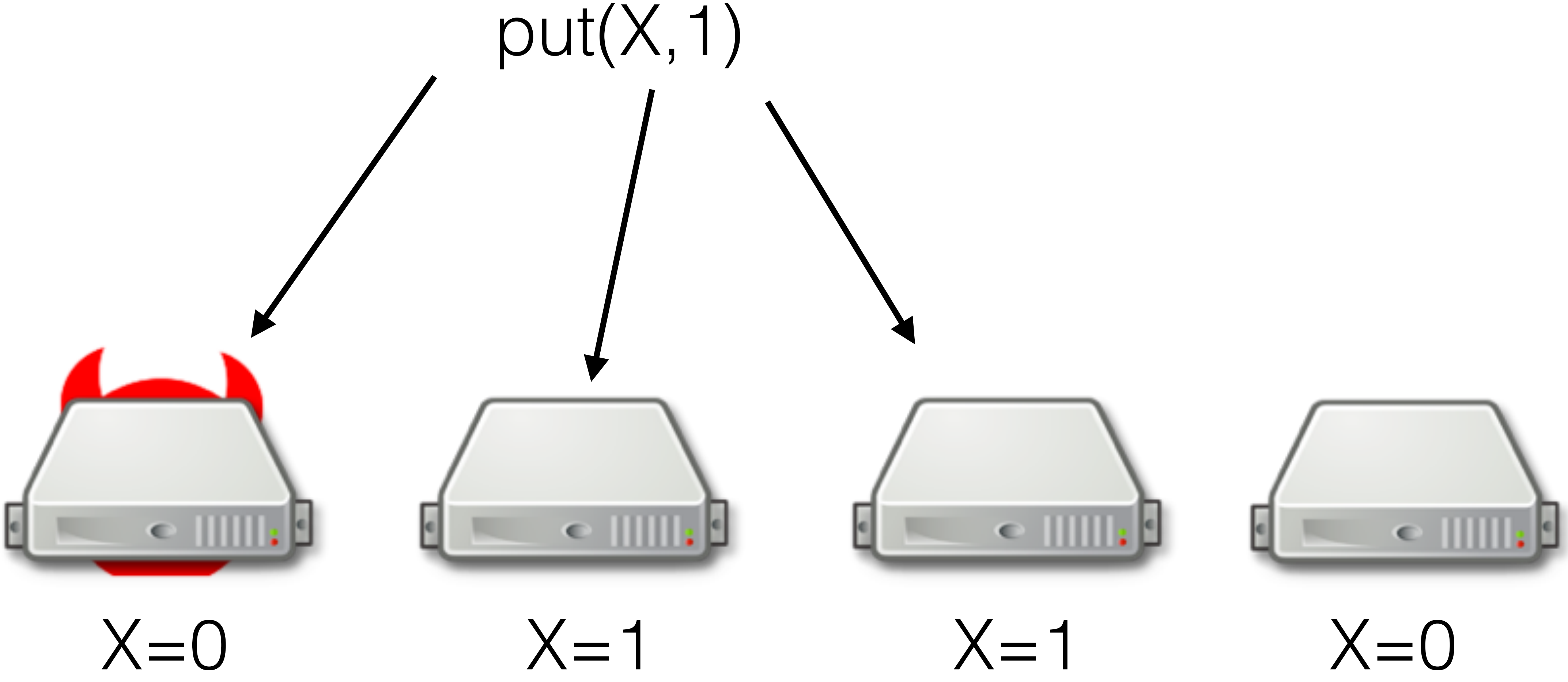
- What if we tried to tolerate Byzantine failures with 2f+1 replicas?



Quorums

- In Paxos: quorums of f+1 out of 2f+1 nodes
  - quorum intersection:
    any two quorums intersect at at least one node
- For BFT: quorums of 2f+1 out of 3f+1 nodes
  - quorum ***majority***
    any two quorums intersect at *a majority* of nodes
    =>
    any two quorums intersect at at least one good node

Are quorums enough?



Are quorums enough?

- We saw this problem before with Paxos:
  just writing to a quorum wasn't enough

- Solution, in Paxos terms:
  - use a two-phase protocol: propose, then accept

- Solution, in VR terms:
  - designate one replica as the primary, have it determine request order
  - primary proposes operation, waits for quorum
    (prepare / prepareOK   = Paxos's accept/acceptOK)

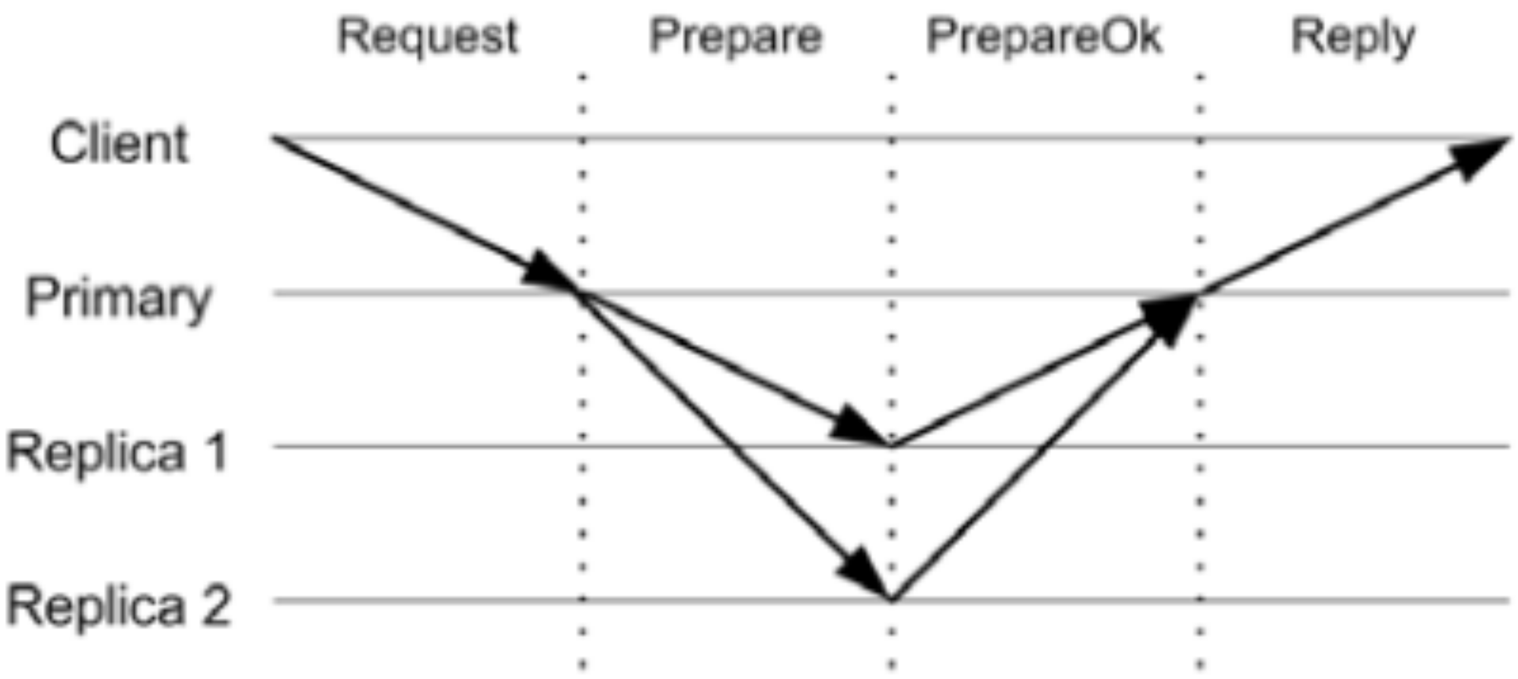
BFT approach

- Use a primary to order requests
- But the primary might be faulty
  - could send wrong result to client
  - could ignore client request entirely
  - could send different op to different replicas
    (this is the really hard case!)

BFT approach

- All replicas send replies directly to client
- Replicas exchange information about ops received from primary
  (to make sure the primary isn’t equivocating)
- Clients notify all replicas of ops, not just primary;
  if no progress, they replace primary
- All messages cryptographically signed

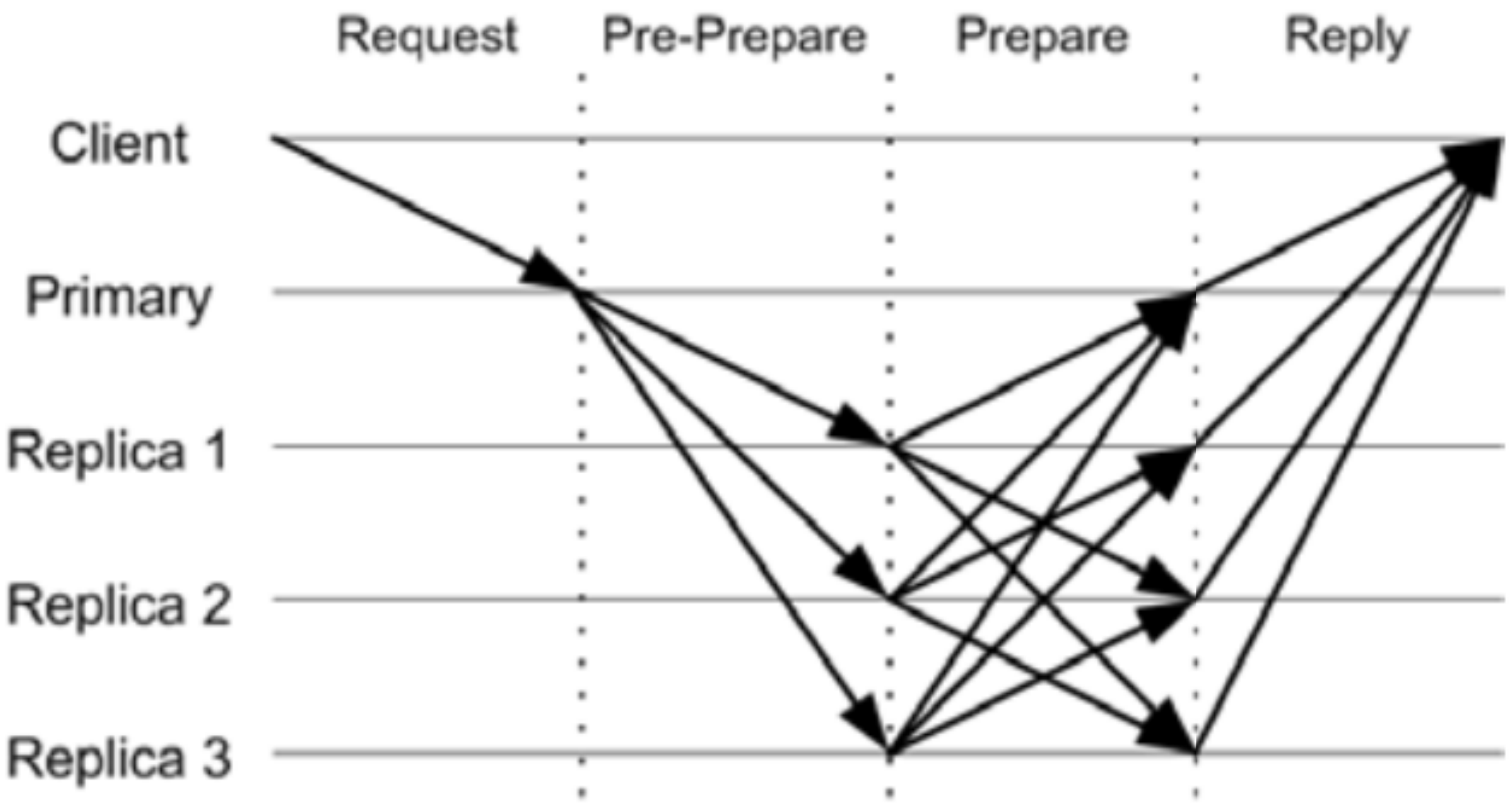
Starting point: VR



- What’s the problem with using this?
  - primary might send different op order to replicas

Next try

- Client sends request to primary & other replicas
- Primary assigns seq number, sends PRE-PREPARE(seq, op) to all replicas
- When replica receives PRE-PREPARE, sends PREPARE(seq, op) to others
  - Once a replica receives 2f+1 matching PREPARES, execute the request



- Can a faulty non-primary replica prevent progress?
- Can a faulty primary cause a problem that won't be detected?
    - What if it sends ops in a different order to different replicas?

Faulty primary

- What if the primary sends different ops to different replicas?
  - case 1: all good nodes get 2f+1 matching prepares
    - they must have gotten the same op
  - case 2: >= f+1 good nodes get 2f+1 matching prepares
    - they must have gotten the same op
    - what about the other (f or less) good nodes?
  - case 3: < f+1 good nodes get 2f+1 matching prepares
    - system is stuck, doesn’t execute any request

View changes

- What if a replica suspects the primary of being faulty? e.g., heard request but not PRE-PREPARE
- Can it start a view change on its own?
  - no - need f+1 requests
- Who will be the next primary?
  - How do we keep a malicious node from making sure it's always the next primary?
  - primary = view number mod n

Straw-man view change

- Replica suspects the primary, sends VIEW-CHANGE to the next primary
- Once primary receives 2f+1 VIEW-CHANGEs, announces view with NEW-VIEW message
  - includes copies of the VIEW-CHANGES
  - starts numbering new operations at last seq number it saw + 1

What goes wrong?

- Some replica saw 2f+1 PREPAREs for op n, executed it
- The new primary did not
- New primary starts numbering new requests at n => two different ops with seq num n!

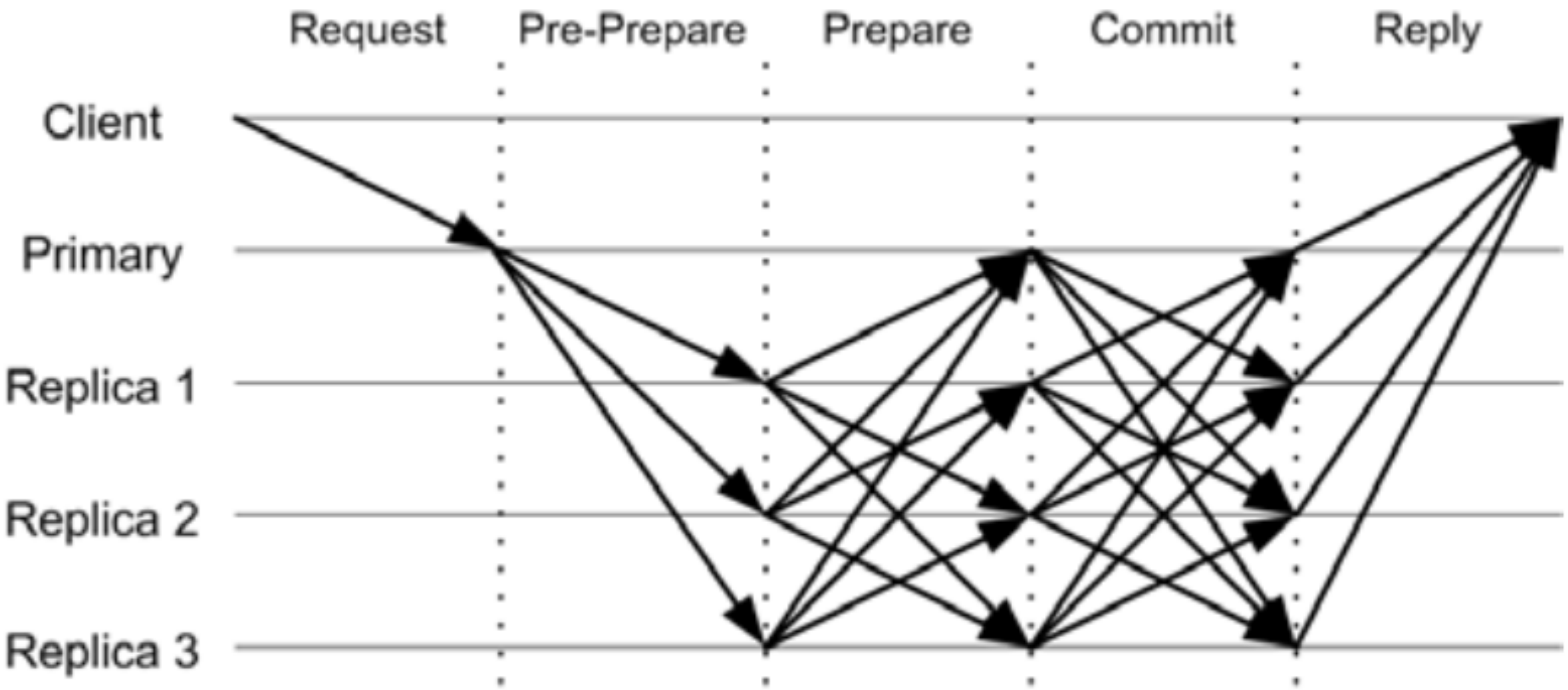
Fixing view changes

- Need another round in the operation protocol!
- Not just enough to know that primary proposed op n, need to make sure that the next primary will hear about it
- After receiving 2f+1 PREPAREs, replicas send COMMIT message to let the others know
- Only execute requests after receiving 2f+1 COMMITs

The final protocol

- client sends op to primary
- primary sends PRE-PREPARE(seq, op) to all
- all send PREPARE(seq, op) to all
- after replica receives 2f+1 matching PREPARE(seq, op),
  send COMMIT(seq, op) to all
- after receiving 2f+1 matching COMMIT(seq, op), execute op, reply to client

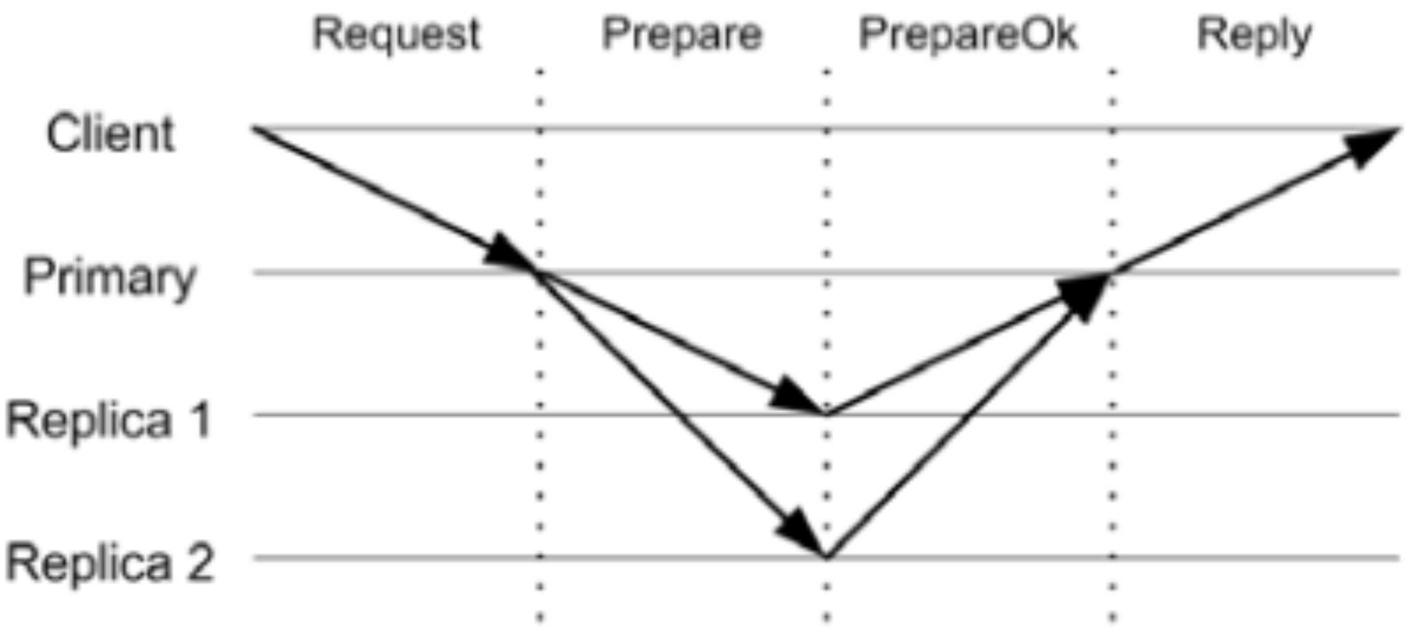
The final protocol

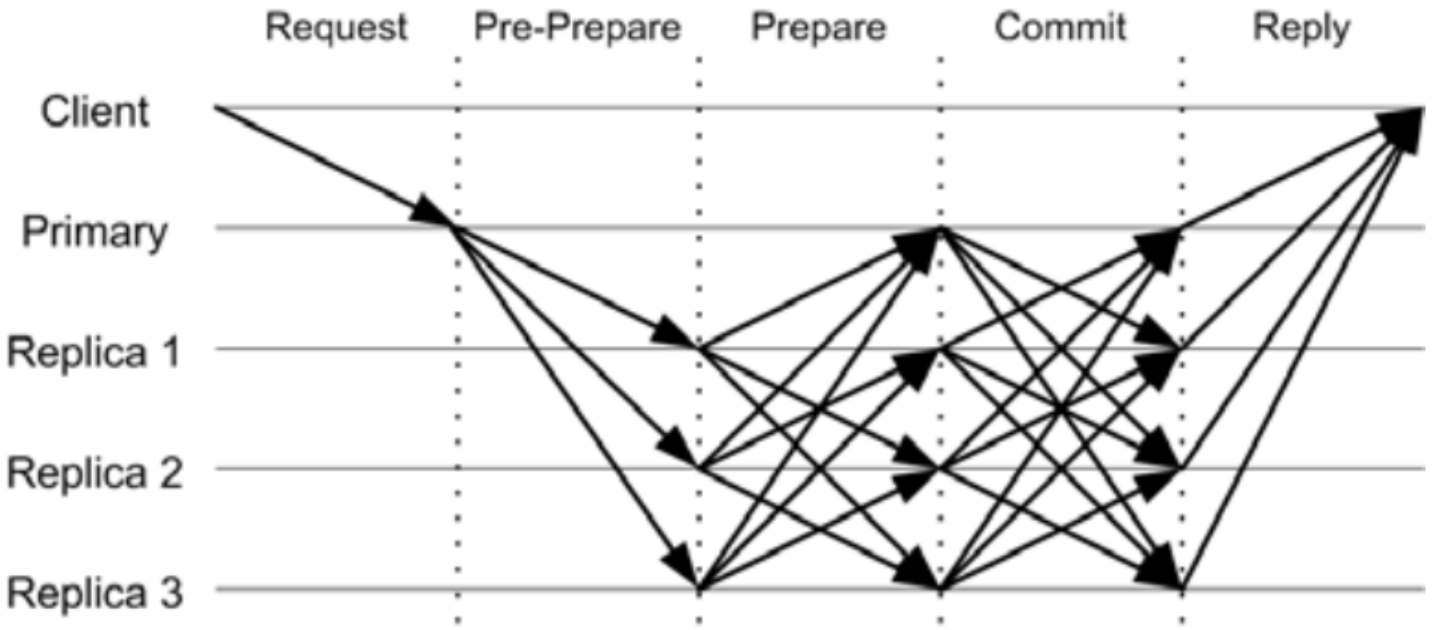


BFT vs VR/Paxos

- BFT: 4 phases
  - PRE-PREPARE - primary determines request order
  - PREPARE - replicas make sure primary told them same order
  - COMMIT - replicas ensure that a quorum knows about the order
  - execute and reply
- VR: 3 phases
  - PREPARE - primary determines request order
  - PREPARE-OK - replicas ensure that a quorum knows about the order
  - execute and reply

BFT vs VR/Paxos





What did this buy us?

- Before, we could only tolerate fail-stop failures with replication
- Now we can tolerate *any* failure, benign or malicious
  - as long as it only affects less than 1/3 replicas
  - (what if more than 1/3 replicas are faulty?)

BFT Impact

- This is a powerful algorithm
- As far as I know, it is not yet being used in industry
- Why?

Performance

- Why would we expect BFT to be slow?
  - latency (extra round)
  - message complexity (O(n^2) communication)
  - crypto ops are slow!

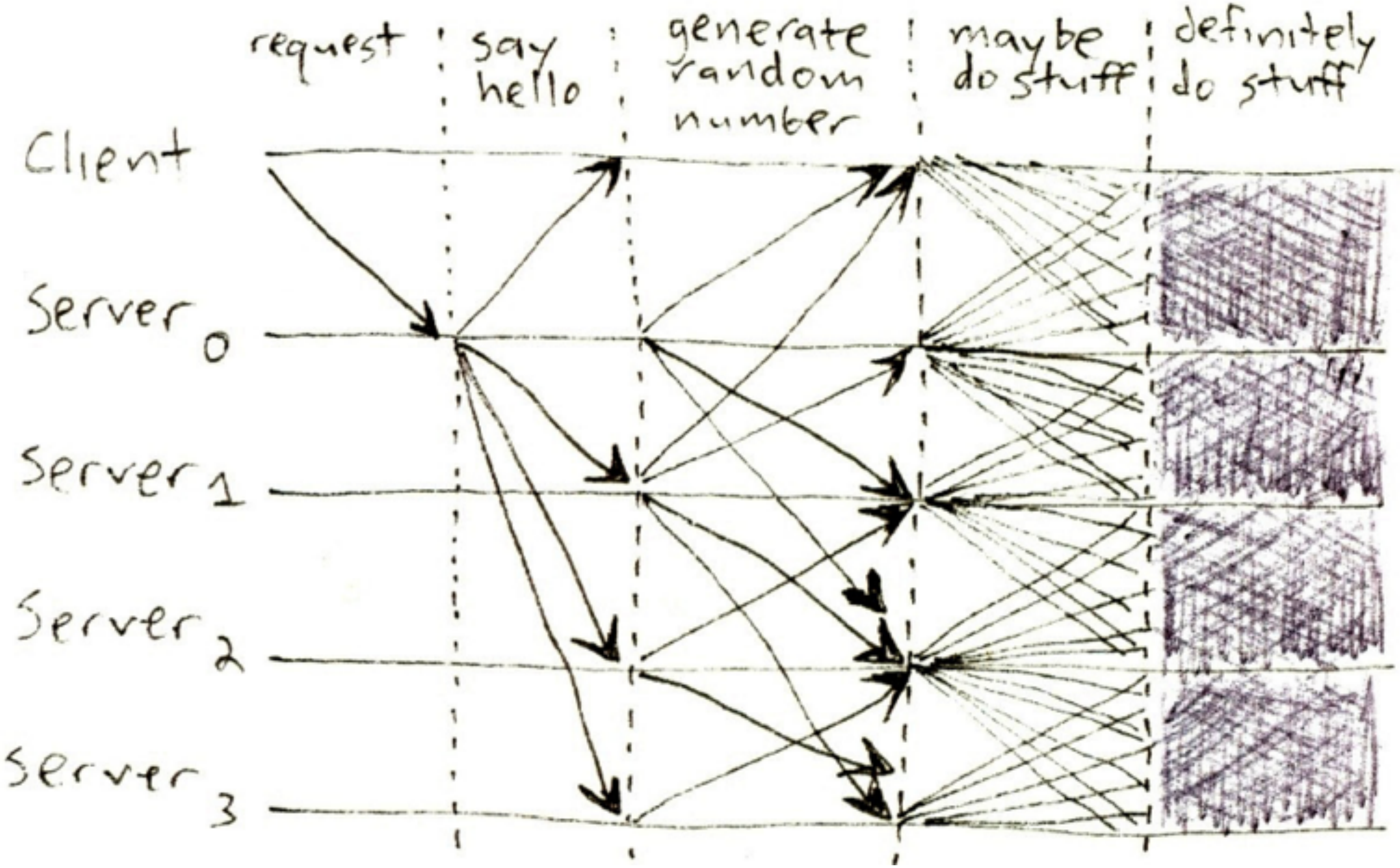
Benchmarks

- PBFT paper says they implemented a NFS file server, got ~3% overhead

- But: NFS server writes to disk synchronously, PBFT only does replication
  (is this ok? fair?)

- Andrew benchmark w/ single client
  => only measures increased latency, not cost of crypto

Implementation Complexity



[J. Mickens, “The Saddest Moment”, 2013]

Implementation Complexity

- Building a bug-free Paxos is hard!
- BFT is much more complicated
- Which is more likely?
  - bugs caused by the BFT implementation
  - the bugs that BFT is meant to avoid

BFT summary

- It’s possible to build systems that work correctly even though parts may be malicious!
- Requires a lot of complex and expensive mechanisms
- On the boundary of practicality?

Bitcoin

- Goal: have an online currency with the properties we like about cash
  - portable
  - can’t spend twice
  - can’t repudiate after payment
  - no trusted third party
  - anonymous

Why not credit cards?

- (or paypal, etc)
- needs a trusted third party which can
    - track your purchases
    - prohibit some actions

Bitcoin

- e-currency without a trusted central party
- What’s hard technically?
    - forgery
    - double-spending
    - theft

Basic Bitcoin model

- a network of bitcoin servers (peers) run by volunteers
  - not trusted; some may be corrupt!
- Each server knows about all bitcoins and transactions
- Transaction (sender —> receiver)
  - sender sends transaction info to some peers
  - peers flood to other peers
  - receiver checks that lots of peers have seen transaction
  - receiver checks for double-spending

Transaction chains

- Every bitcoin has a chain of transaction records
  - one for each time it's been transferred
- Each record contains
  - public key of new owner
  - hash of this bitcoin's previous transaction record
  - signed by private key of old owner
  - (in reality: also fractional amounts, multiple recipients, …)

Example

- Bob has a bitcoin received from Alice in T7
    - T7: pub(Bob), hash(T6), sig(Alice)
- wants to buy a hamburger from Charlie
    - gets his public key
    - creates T8: pub(Charlie), hash(T7), sig(Bob)
    - sends transaction to Bitcoin peers to store
    - Charlie verifies that the network has accepted T8, gives Bob the hamburger

Stealing

- Does this approach prevent stealing someone else's bitcoins?
- Need a user's private key to spend a coin
- Challenge: what if an attacker steals Bob's private key?
  - significant problem in practice!

Double-Spending

- Does this design so far prevent double-spending?
- What keeps Bob from creating two different transactions spending the same bitcoin?
- Need to make sure the bitcoin peers properly verify a transaction:
    - T8’s signature matches T7’s pub key
    - there was no prior transaction that mentioned hash(T7)

Verifying the transaction chain

- Need to ensure that every client sees a consistent set of operations
    - everyone agrees on which transactions happened and in what order

- Could achieve with a central server maintaining a log, but we wanted to avoid that!

Can we use BFT?

- In theory, yes, but…
- BFT does not scale to large numbers of replicas!
- Can we ensure that malicious nodes make up less than 1/3rd of the replicas?

Sybil attacks

- You can have as many identities as you want on the internet!
- So an attacker could run many replicas, overwhelm the honest nodes
  (limited only by network bandwidth, etc)
- How does BFT deal with this problem?
- How does Bitcoin deal with this problem?

The blockchain

- Full copy of all transactions stored in each peer
- Each block:
  hash(previousblock), set of transactions, nonce
- Hash chain implies order of blocks
- A transaction isn't real until it's in the blockchain

Extending the blockchain

- How do peers add to the blockchain?
- All the peers look at the longest chain of blocks,
  try to create a new block extending the previous block
- Requirement: hash(new block) < target
  - peers must find a nonce value that works by brute force
  - requires months of CPU time, but thousands of peers
    are working on it => new block every 10 minutes
- when new block created, announce it to all peers

Proof of work

- Why do peers have to work to find correct nonces?
- This solves the sybil attack problem
  without a central authority or admission control
  - BFT required less than 1/3 replicas faulty
  - Bitcoin requires less than 1/2 *the CPU power*
    controlled by faulty replicas
    (actually, some attacks possible if 1/3 faulty)

Double-spending

- Start with blockchain …->B6
- Bob creates transaction B->C, gets it into blockchain
  … -> B6 -> B7, where B7 contains B->C
  - so Charlie gives him a hamburger
- Can Bob create another block Bx and get peers to accept chain … -> B6 -> Bx instead?

Double-spending

- When will a peer accept a new chain it hears about?
    - When it’s longer than all other chains it’s seen
- So an attacker needs to produce a longer chain to double-spend
    - needs to create B6->Bx->B8, longer than B6->B7
    - and needs to do that before the rest of the network creates a new block (10 minutes)
    - so the attacker needs to have more CPU power than the rest of the network

Bitcoin summary

- Building a peer-to-peer currency involves lots of technical problems:
  preventing theft, double-spending, forgery even though some participants may be malicious

- Using CPU proof-of-work instead of BFT-like protocol avoids Sybil attacks

- Also lots of non-technical problems:
  why does it have value, legality?

Wrapup

- What have we learned?

From the first lecture:

We want to build distributed systems to be more scalable, and more reliable.

But it’s easy to make a distributed system that’s *less scalable and less reliable* than a centralized one!

Distributed Systems Challenges

- Managing communication
- Tolerating partial failures
- Keeping data consistent
  despite many copies and massive concurrency
- Scale and performance requirements
- Malicious behavior
- Testing

We’ve seen a variety of tools for addressing these challenges

- Managing communication: RPC and DSM
- Tolerating failures:
  Paxos, VR, Chain Replication, NOPaxos
- Keeping data consistent:
  replication, transactions, cache coherency
- Scale and performance:
  partitioning, caching, consistent hashing
- Security: BFT
- Testing: model checking and verification

We’ve seen how these are used in various real systems

- The Google storage stack:
  GFS, Chubby, Bigtable, Megastore, Spanner

- Weak consistency systems:
  Amazon’s Dynamo, COPS

- Data analytics:
  MapReduce, GraphLab, Spark

We’ve *built* systems that solve these problems

- Fault-tolerant MapReduce (Lab 1)
- Fault tolerant state through Paxos/replication (Lab 2/3)
- Scalability through sharding (Lab 4)

- Building a replicated sharded key-value store
  is a major accomplishment!

- **Lesson:** know when to use these design patterns to solve distributed systems challenges

- Many of the systems we looked at use:
  RPC, state machine replication, Paxos, transactions…

- Reuse these algorithms even if not code

- **Lesson:** know when to avoid solving hard problems you don’t need to

- Example: MapReduce loses data on certain failures; GFS uses a centralized, in-memory master

- **Lesson:** recognize and avoid trying to solve impossible problems

- Example: can’t guarantee consistency and perfect availability and low latency in all cases, so use eventual consistency when this matters (Dynamo)
- Example: can’t make failures completely transparent with RPC

Distributed Systems are Exciting!

- Some of the hardest challenges we face in CS
- Some of the most powerful things we can build
    - systems that span the world,
      serve millions of users,
      and are always up