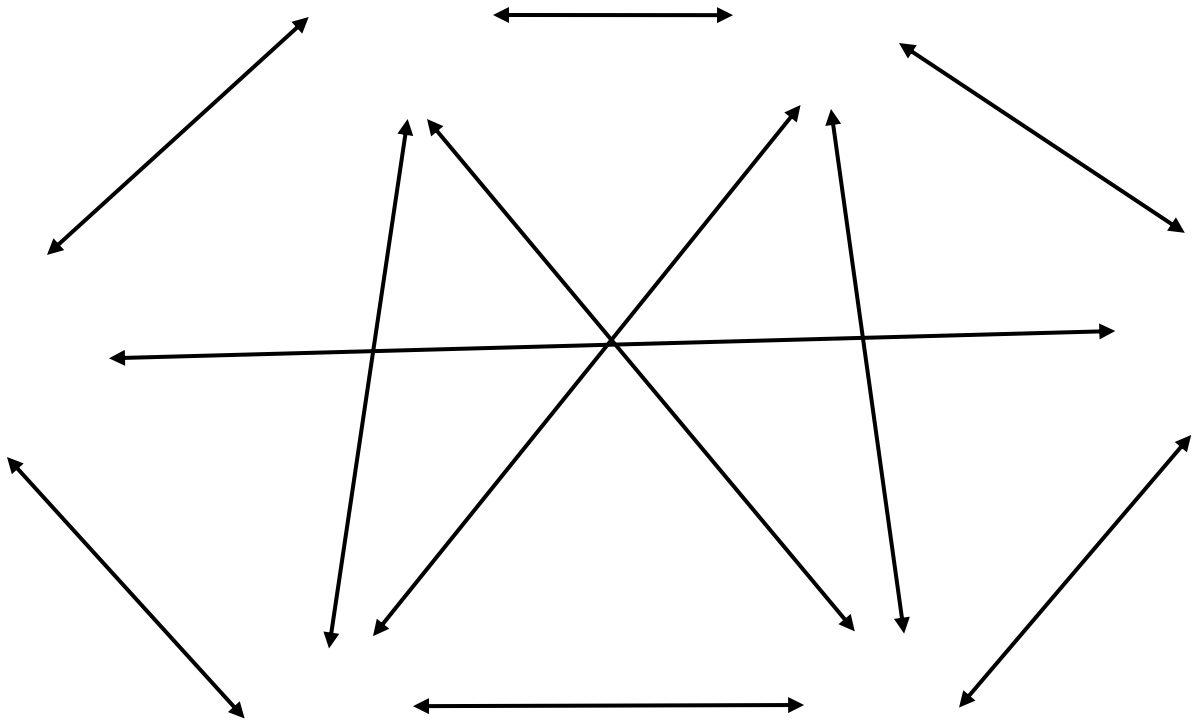
Consensus and Byzantine Failures

CS 839 – Kai Mast

Project Timeline

- October 8th: Proposal Due
- November 5th: Related Work Due
- Before December 1st: Final Check-in
- December 15th: Project Presentations
- December 20th: Project Reports Due

Recap: Blockchains and Consensus



Blockchains are Distributed Ledgers

- Ledgers are records of transactions
- Copies of the ledger exist on many different machines (or nodes)
- Nodes might have benign failures (crashes, disconnects, power outages,…)
- Nodes may also try to modify the ledger to their advantage

Recap: Failure Models

Crash Failures (Paxos, Raft, etc.):

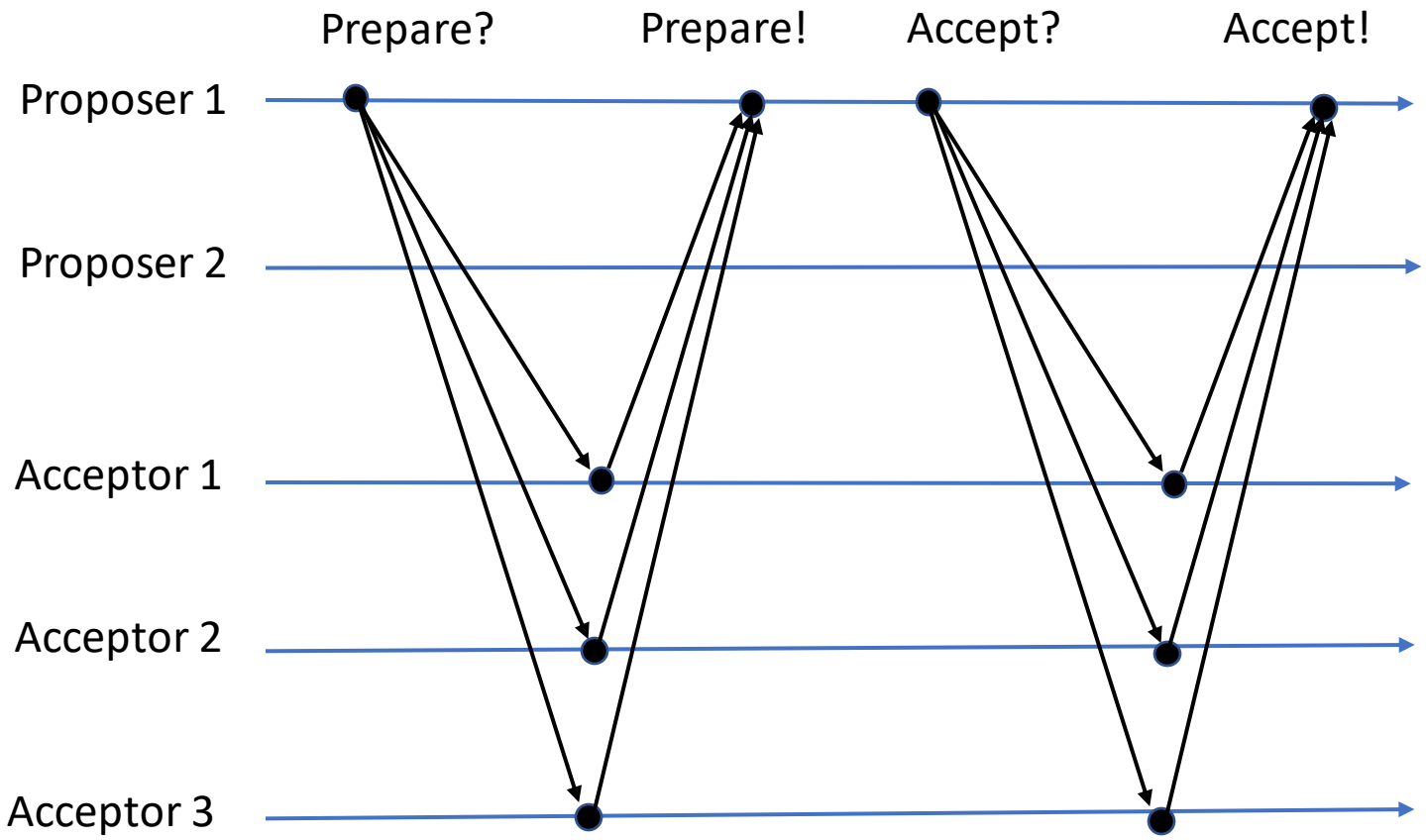
- Nodes fail without side-effects
- Failures cannot be detected reliably

Byzantine Failures (PBFT and friends):

- Nodes can have arbitrary bugs
- Nodes can intentionally misbehave or lie

Byzantine Failures are a **superset** of Crash Failures!

Recap: Paxos



Phase 1: Prepare

- Elect a leading/primary proposer
- Find out about any (potentially) chosen values
- Block older proposals that have not yet completed

Phase 2: Accept

- Primary asks acceptors to accept a specific value

Correctness Properties of Distributed Systems

- **Safety:** "Nothing bad will happen"
  - Here, the majority of the system agrees on the same sequence of transactions
- **Liveness:** "Something will happen eventually"
  - Here, the system will accept a transaction in some finite amount of time

Practical Byzantine Fault Tolerance (PBFT)

- First efficient solution to the Byzantine Generals problem
- Published in SOSP 1999

Miguel Castro

- In the 90s/early 2000s: mostly intended to handle complex benign failures (e.g., software bugs)
- Today: Powers many permissioned blockchains

Barbara Liskov

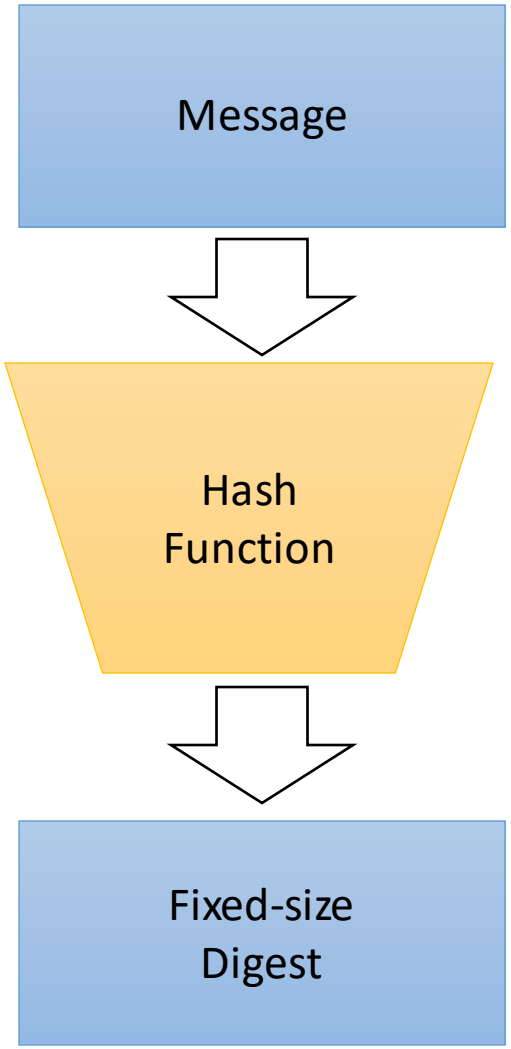
Disclaimer: This is probably one of the hardest papers we will read

Why Practical?

- A Byzantine fault-tolerant consensus protocol has been proposed before
  - See "The Byzantine Generals Problem" (Lamport 1982)
  - But computationally expensive
- PBFT is comparatively efficient

Cryptographic Primitives: Hash Functions

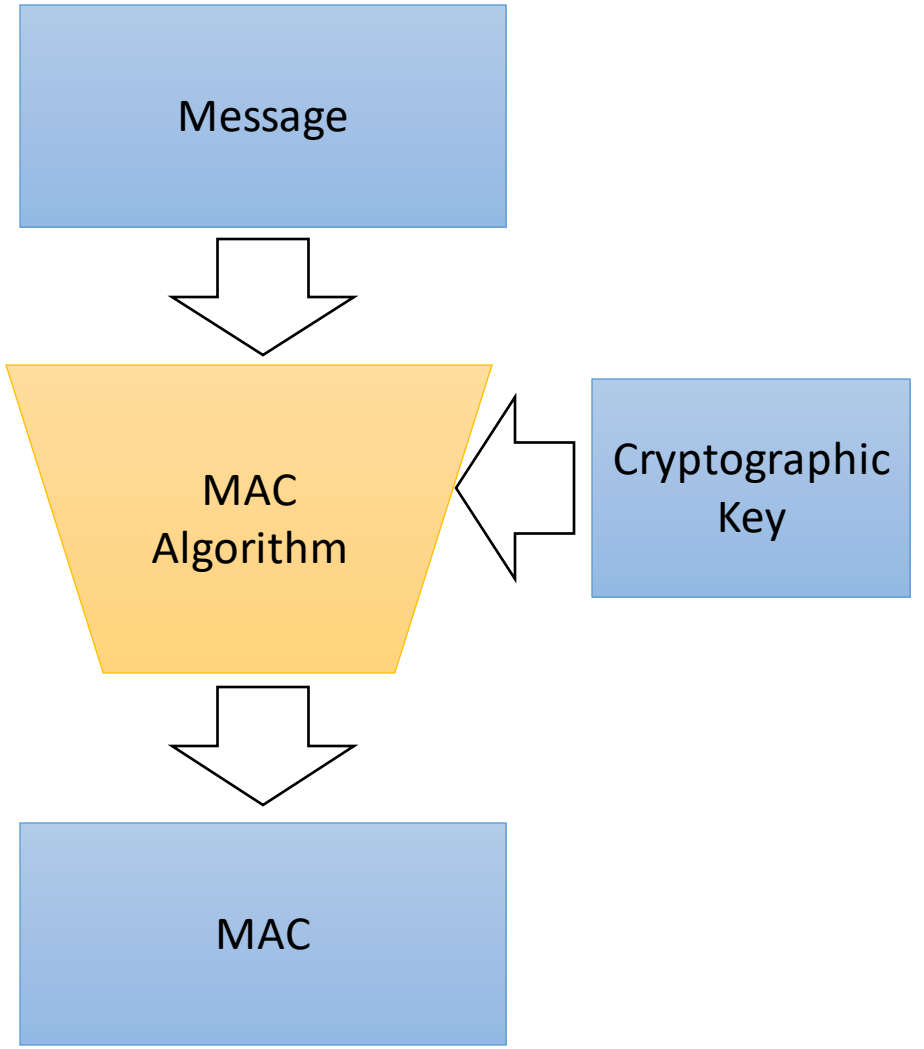
- Convert an arbitrarily large message into a succinct representation
- Hash functions are used in PBFT to verify the **integrity** of a message
- Cryptographic hash functions make it virtually impossible to find two messages that match the same digest
  - Why is this important?



Cryptography: Message-Authentication Codes

- MACs allow verifying the **authenticity** and **integrity** of a message
- Requires to negotiate a key between the prover and verifier

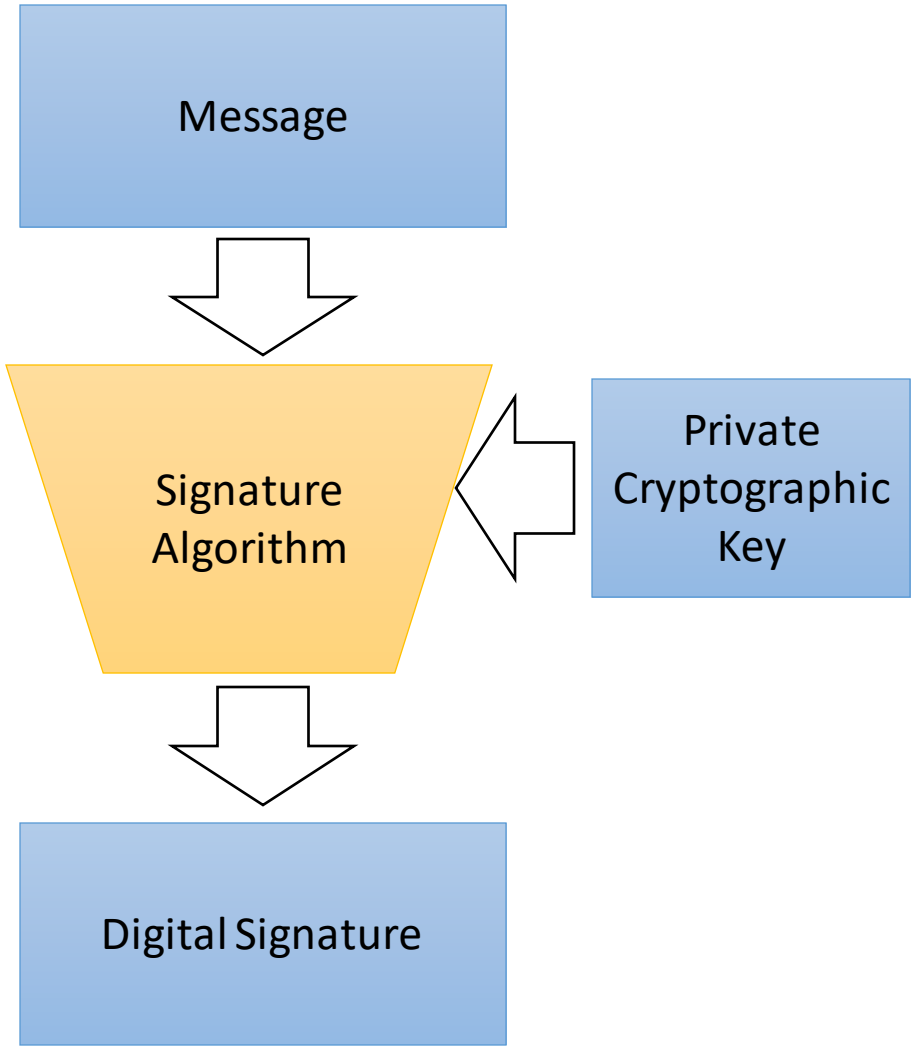
- What is the difference to encryption?



Cryptography: Digital Signatures

- Digital signatures allow to verify the origin/authenticity of a message
- Like MACs, they also allow verifying integrity
- Unliked MACs, digital signatures are asymmetric
  - Private key to sign
  - Public key to verify signatures

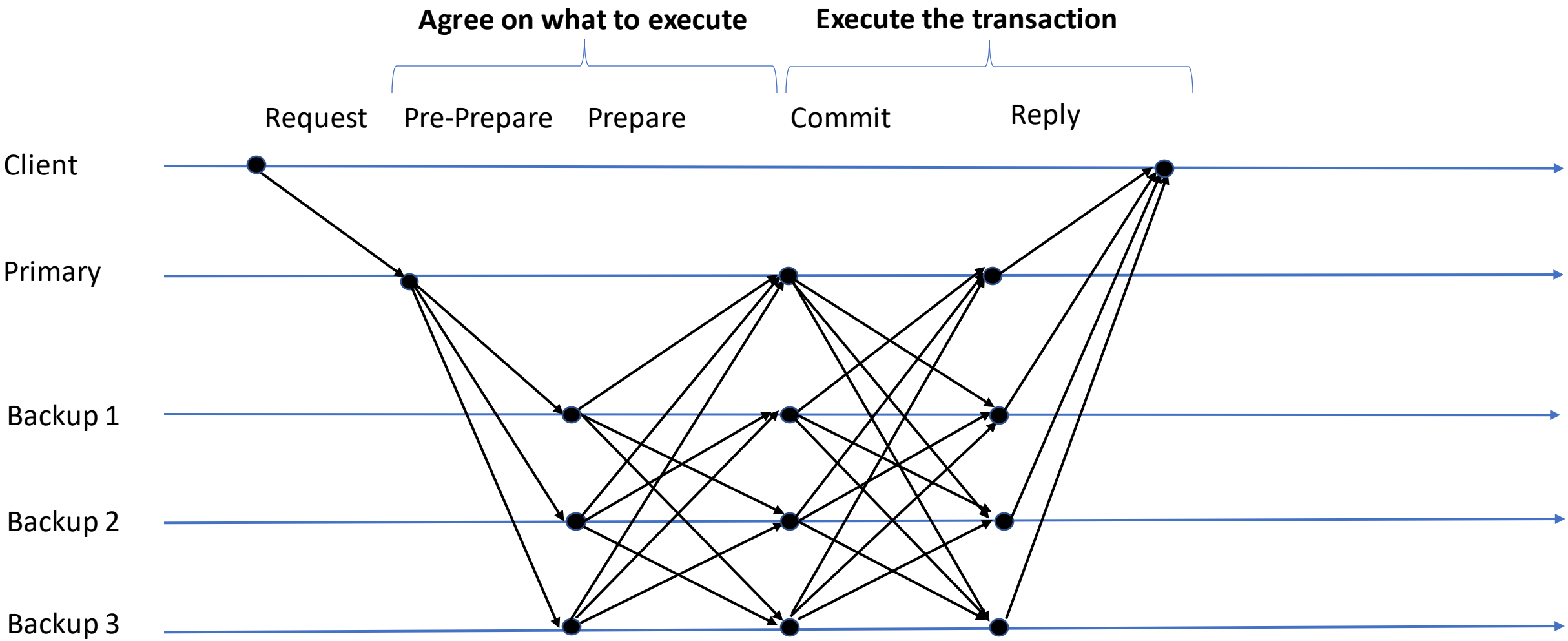
- Asymmetric encryption and signatures are **much more** expensive



PBFT in a nutshell

- Replicas are either the primary or a backup
- All participants connect to all other participants and negotiate a MAC key for each connection
- Transaction flow:
    1. Clients send a transaction request to primary.
    2. Primary forwards request to all backups.
    3. Replicas agree on whether to execute the transactions
    4. Replicas execute the request and send the reply to the client
    5. Client waits for f + 1 responses with the same

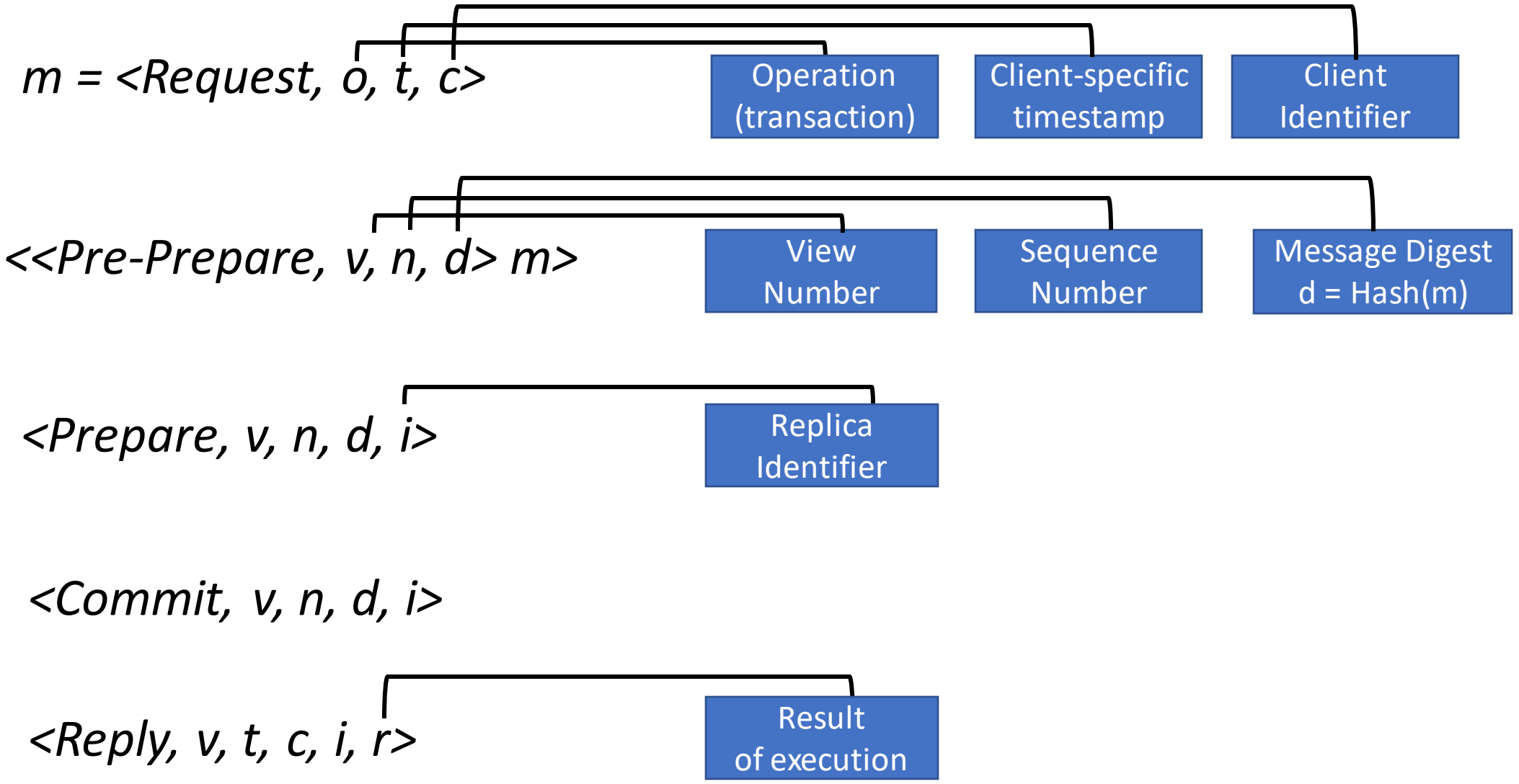
PBFT in a nutshell



PBFT Views

- Each view has a single primary
- Faulty primaries can be replaced using a **view change**
- Primaries can fail by
  - Timing out
  - Sending invalid messages
  - Sending conflicting messages

PBFT Messages in Detail



How many replicas are needed?

- Paxos: 2f+1
- PBFT: 3f+1

Why?

- Must be able to proceed after getting n-f responses
- But faulty nodes might respond faster than non-fault nodes

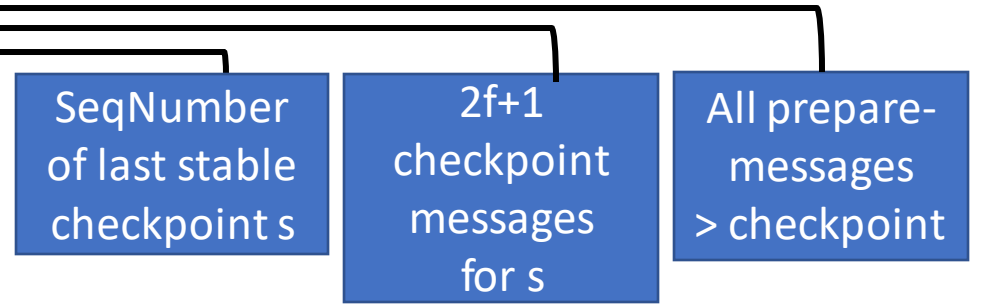
Message Log

- Replicas keep track of messages, so that
  - Other machines can recover state
  - We can recover state during view change

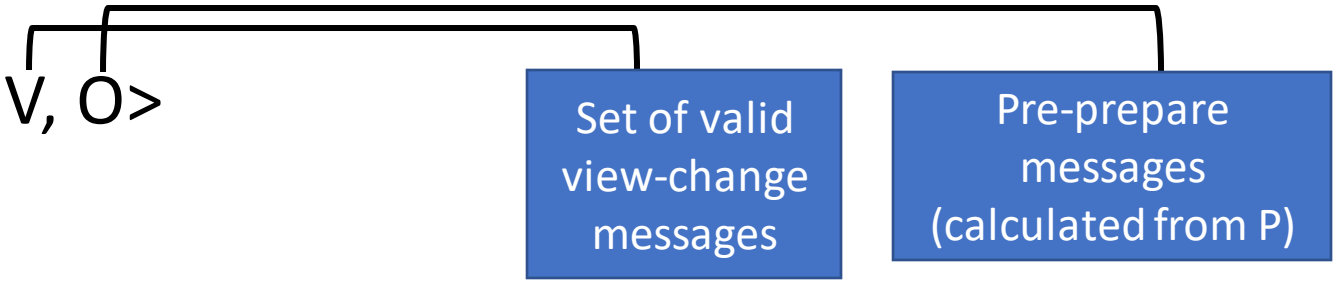
- Checkpoints allow
  - Discarding old messages in the log
  - Create a self-standing state certificate that can be verified against public keys

View Changes

- Upon detecting a primary failure, backups start a view change
  - Broadcasts <View-Change, v+1, n, C, P, i>
  - Stops accepting messages for view v



- When the primary for view v+1 receives 2f+1 valid view change messages
  - Broadcasts <New-View, v+1, V, O>



Safety: Can we commit two conflicting transactions?

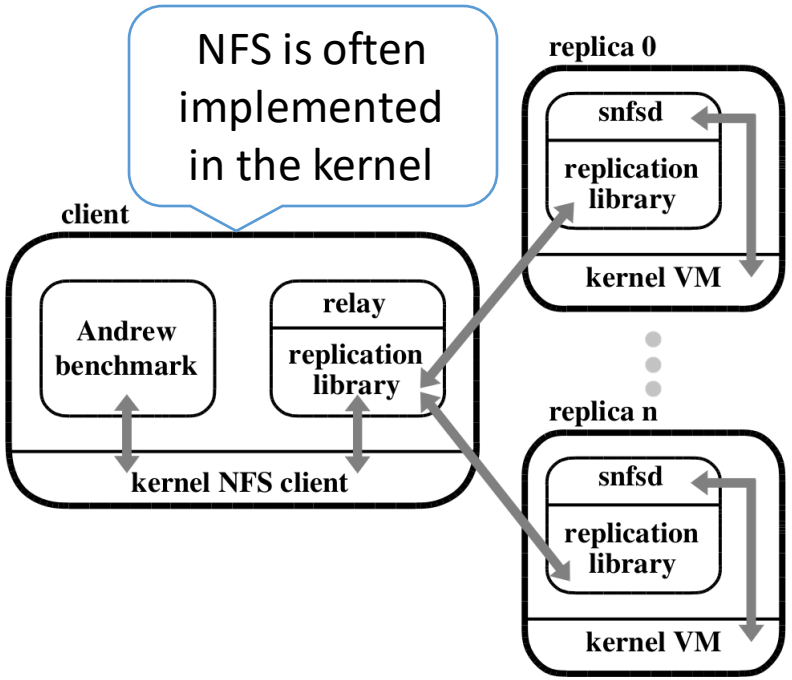
1. A correct node will only prepare at most one transaction for any (view_number, sequence_number)-pair
2. A correct node will only prepare transactions that do not violate state
   - E.g., won't accept a double spend
3. A correct node will only issue a commit if it received 2f+1 prepare messages

Liveness: What if multiple replicas fail at once?

- Nodes will switch views again if no response after some timeout delay
- Timeout delay grows exponentially with every unsuccessful view change

View changes are the most difficult part of the protocol. Can make or break its safety and liveness guarantees.

Evaluation: An NFS server



Different workloads for the fs benchmark

BFS with two different fs semantics

Baseline without replication

phase	BFS strict	BFS r/o lookup	BFS-nr
1	0.55 (57%)	0.47 (34%)	0.35
2	9.24 (82%)	7.91 (56%)	5.08
3	7.24 (18%)	6.45 (6%)	6.11
4	8.77 (18%)	7.87 (6%)	7.41
5	38.68 (20%)	38.38 (19%)	32.12
total	64.48 (26%)	61.07 (20%)	51.07

Execution times in seconds
(percentages are changes w.r.t. baseline)

Discussion: Evaluation sufficient?

- Would be great to see performance a function of replicas
  - Benchmarks only consider n=4 and f=1
- How do failures affect performance?
- What about other applications

But:

- First system of its kind. Proof it works is sufficient.

Discussion: Independent failures realistic?

- It's hard to have 2f+1 *fully* independent implementation
  - Implementations may rely on the same (faulty) specification
  - Bugs might be copied even across languages (especially if they have similar syntax)

- In the blockchain setting adversaries might collude or try to exploit common software bugs
  - Here, we also need to take economic incentives into account!

Optimization: Batching

- Modern implementations of PBFT (and similar protocols) do not agree on singular transactions
- Instead, they bundle hundreds (sometimes thousands) of transactions into one batch

- Increases throughput significantly but:
  - May increase end-to-end latency
  - Primary can reorder transactions inside the batch to their advantage ("front-running")

More Optimizations…

- Only primary sends full result
  - Result may be big; can be requested from the backups on faulty primary
- Optimistic commits
  - Commit prepared operations tentatively.
  - Rollback in case of failures
  - Assumption: Byzantine failures are rare

That's it for today

- Next time: Bitcoin and Nakamoto Consensus
- Please sign up for lecture notes
- Feel free to come to office hours!