Lecture #5: Byzantine Faults and Digital Signature Schemes

COMS 4995-001: The Science of Blockchains

URL: https://timroughgarden.org/s25/

Tim Roughgarden

Goals for Lecture #5

1. The challenges of Byzantine faults.
   – faulty validators that can behave in arbitrary (worst-case) ways

2. Digital signature schemes.
   – key tool for limiting the space of Byzantine validator strategies

3. Limits on what is achievable.
   – Byzantine faults make the SMR problem harder in partial synchrony

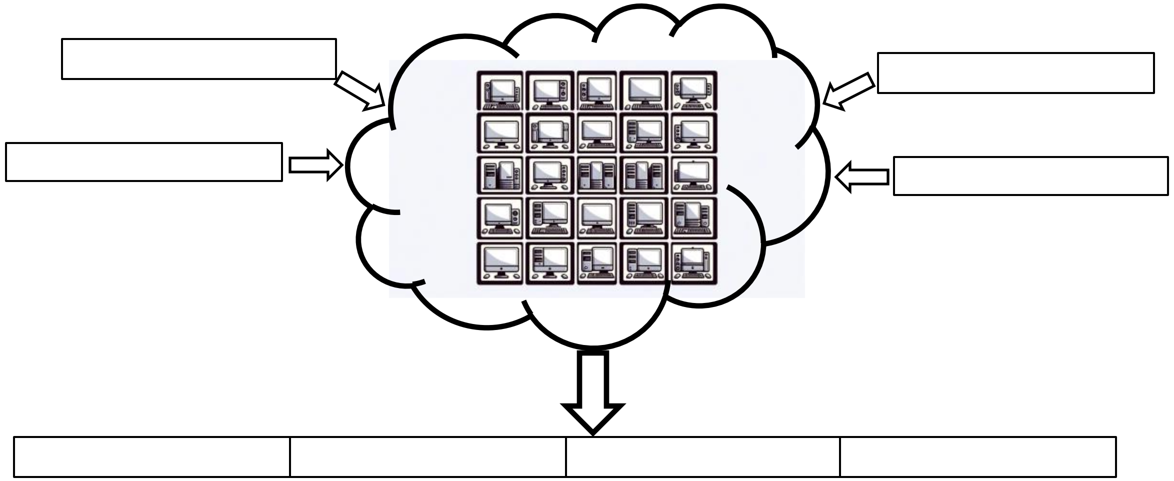
4. Key ideas behind Tendermint.
   – Full protocol description and analysis on Monday.

State Machine Replication (SMR)

SMR: version of consensus appropriate for a blockchain protocol.

- “state machine” = for us, current state of virtual machine
- “replication” = all validators perform same state transitions
- “clients” submit transactions (“txs”) to validators
- each validator maintains an append-only list of finalized txs (a.k.a. “log” or “history”)

Goal: a protocol that satisfies consistency and liveness.



A Road Map to Practical SMR Protocols

crash faults +
synchrony
[security threshold ≈ 100%]

crash faults +
partially synchrony
[security threshold = 50%]

Byzantine faults +
partially synchrony
[security threshold = ?]

easier → harder

Lecture #3: Protocol B solves SMR with crash faults in synchrony.

Lecture #4: if strict majority of validators are non-faulty, Protocol C (≈ Paxos/Raft) solves SMR with crash faults in partial synchrony.

Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults.

- faulty validators can act arbitrarily

Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults.

- faulty validators can act arbitrarily
  - original motivation (1980s): hard-to-model software errors
  - blockchain protocols: might literally get attacked by hostile actor
    - e.g., hacks into validators previously controlled by good actors
  - alternative names for non-faulty validators: “honest,” “correct”

Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults.

- faulty validators can act arbitrarily
  - original motivation (1980s): hard-to-model software errors
  - blockchain protocols: might literally get attacked by hostile actor
    - e.g., hacks into validators previously controlled by good actors
  - alternative names for non-faulty validators: “honest,” “correct”
  - question: what are Byzantine validators capable of?

Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults.

- faulty validators can act arbitrarily
  - original motivation (1980s): hard-to-model software errors
  - blockchain protocols: might literally get attacked by hostile actor
    - e.g., hacks into validators previously controlled by good actors
  - alternative names for non-faulty validators: “honest,” “correct”
  - question: what are Byzantine validators capable of?

Question: is Protocol C (≈ Paxos/Raft) still live and consistent with Byzantine faults?

Paxos/Raft with Byzantine Faults

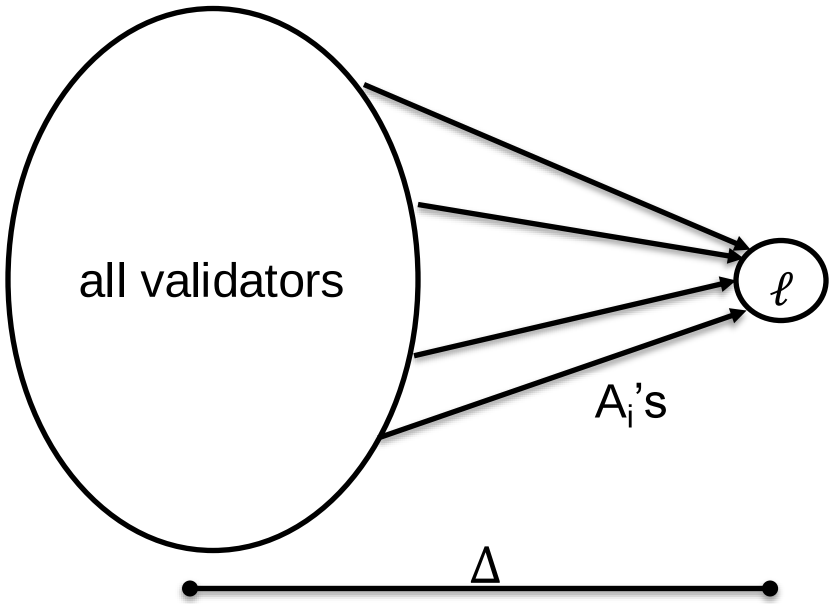
Next challenge: Byzantine faults (i.e., faulty validators can act arbitrarily).

Question: is Protocol C (≈ Paxos/Raft) still live and consistent with Byzantine faults?

Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults (i.e., faulty validators can act arbitrarily).

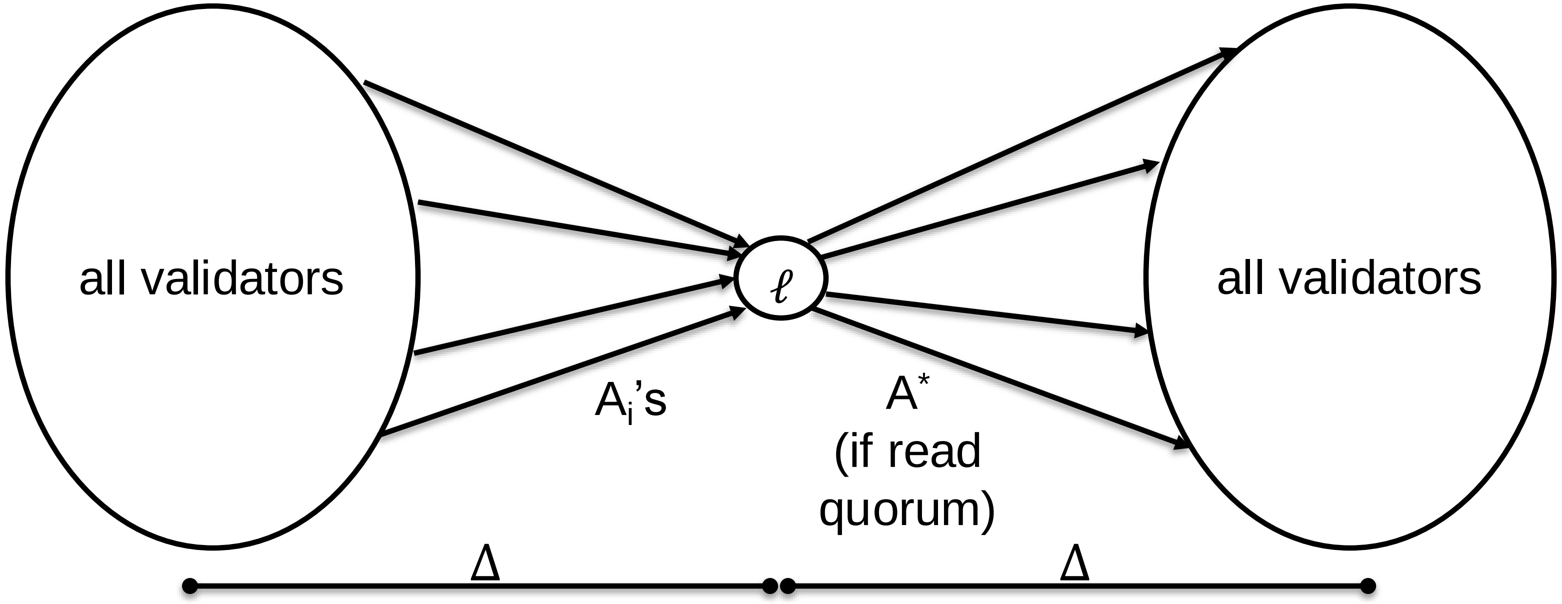
Question: is Protocol C (≈ Paxos/Raft) still live and consistent with Byzantine faults?



Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults (i.e., faulty validators can act arbitrarily).

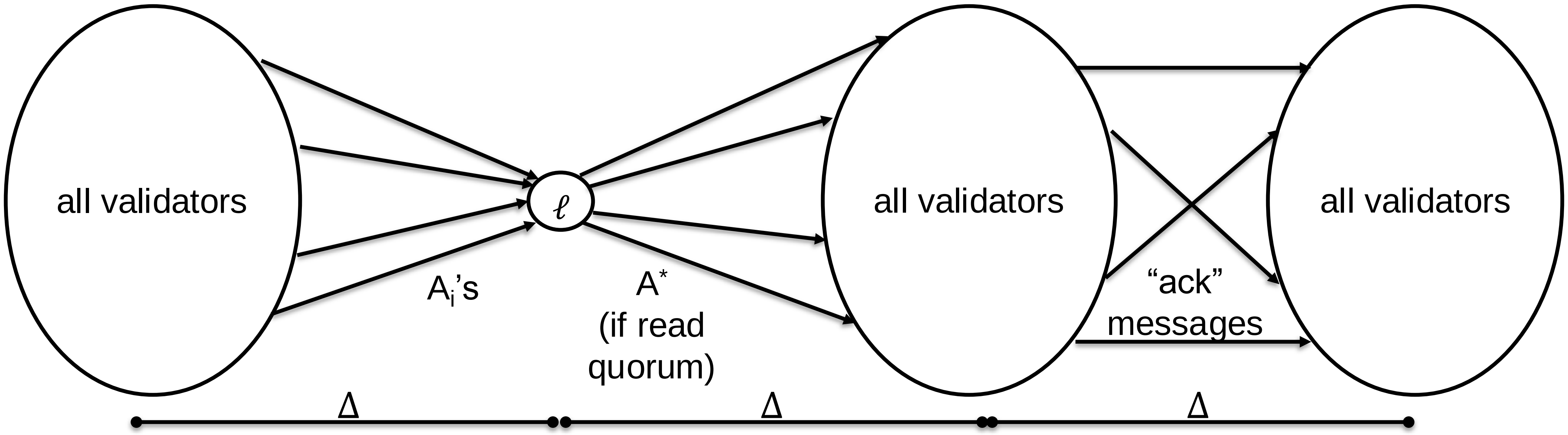
Question: is Protocol C (≈ Paxos/Raft) still live and consistent with Byzantine faults?



Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults (i.e., faulty validators can act arbitrarily).

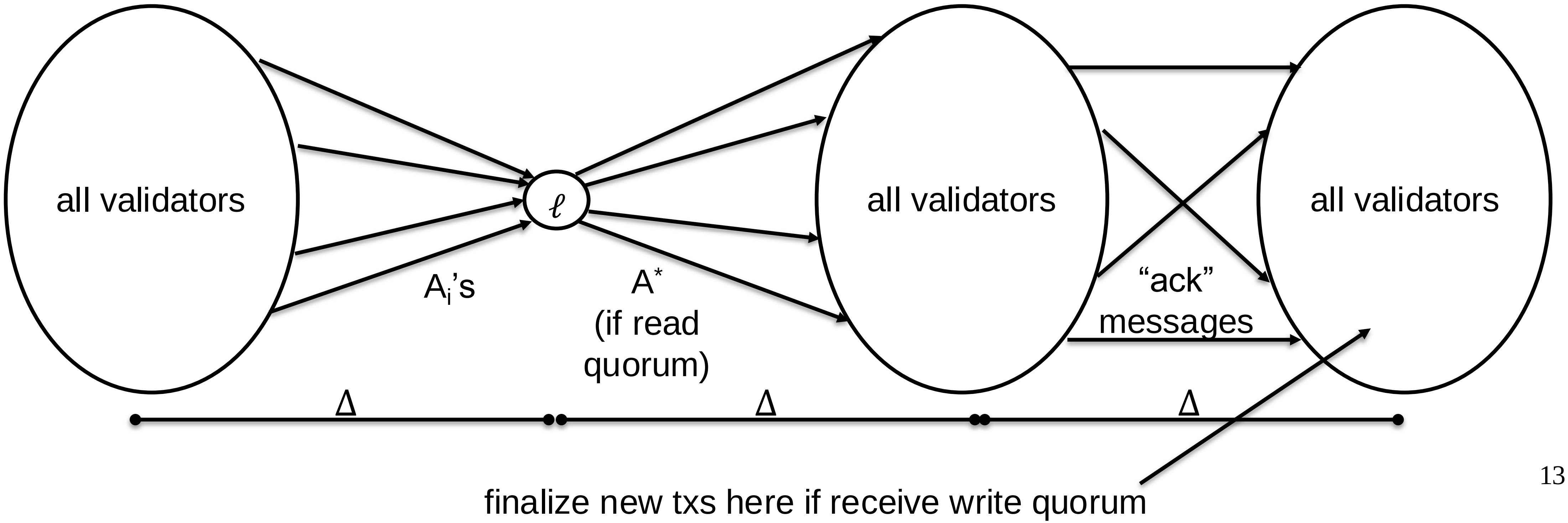
Question: is Protocol C (≈ Paxos/Raft) still live and consistent with Byzantine faults?



Paxos/Raft with Byzantine Faults

Next challenge: Byzantine faults (i.e., faulty validators can act arbitrarily).

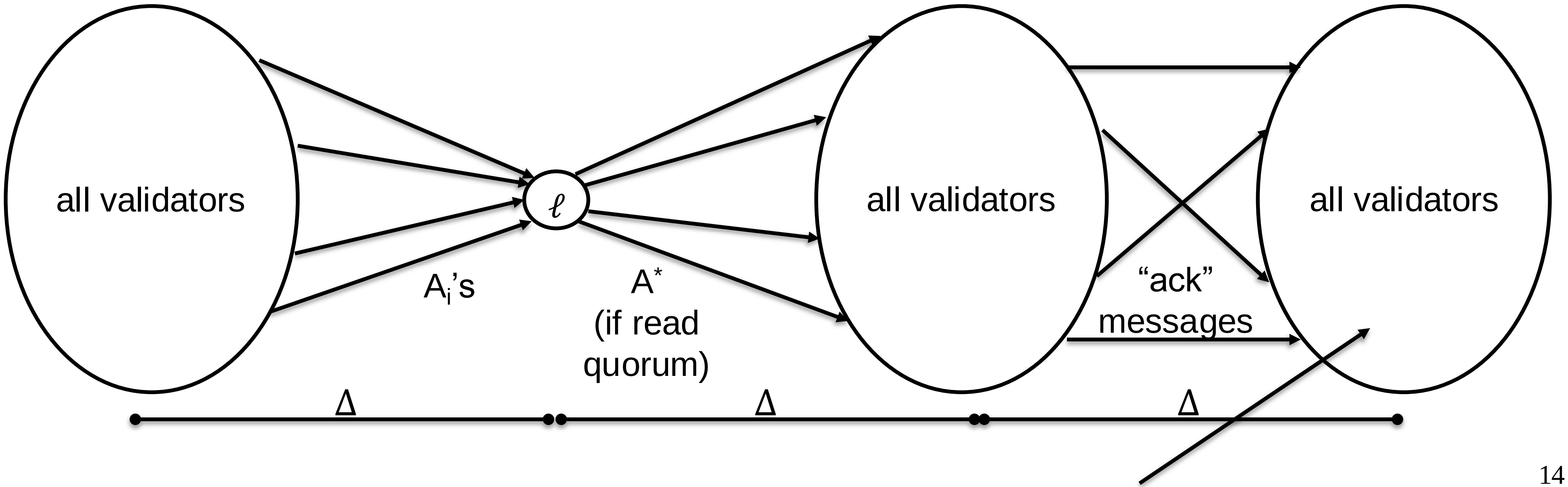
Question: is Protocol C (≈ Paxos/Raft) still live and consistent with Byzantine faults?



Paxos/Raft with Byzantine Faults

Question: is Protocol C still live and consistent with Byzantine faults?

- key property for consistency: read quorum must intersect write quorums from all previous views ➔ if leader makes a proposal, must be up-to-date



finalize new txs here if receive write quorum

Byzantine Validator Shenanigans

Byzantine Validator Shenanigans

Issue #1: Byzantine leader could ignore read quorum requirement and make an (out-of-date) proposal anyway.

- maybe didn't receive chains from > n/2 validators, or maybe it did and chose to ignore them
- out-of-date proposal (if adopted) ➔ consistency violation

Byzantine Validator Shenanigans

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Byzantine Validator Shenanigans

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Issue #2: Byzantine leader could propose different chains to different validators [a.k.a. “equivocation”].

Byzantine Validator Shenanigans

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Issue #2: Byzantine leader could propose different chains to different validators [a.k.a. “equivocation”].

- but can’t only one proposal garner the necessary >n/2 acks?

Byzantine Validator Shenanigans

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Issue #2: Byzantine leader could propose different chains to different validators [a.k.a. "equivocation"].

- but can't only one proposal garner the necessary >n/2 acks?
- no: Byzantine validators can ack multiple proposals
  - non-faulty validators might simultaneous finalize inconsistent chains

Byzantine Validator Shenanigans

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Issue #2: Byzantine leader could propose different chains to different validators, all supported by acks from Byzantine validators. (➔ consistency violation)

Byzantine Validator Shenanigans

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Issue #2: Byzantine leader could propose different chains to different validators, all supported by acks from Byzantine validators. (➔ consistency violation)

Issue #3: Byzantine validators could lie about messages received from other validators.

- e.g., frame a non-faulty validator for its own misbehavior
- will tackle this issue with cryptography (next)

Digital Signature Schemes in Blockchains

- one of the two most ubiquitous cryptographic primitives used in blockchain protocols (along with cryptographic hash functions)

Digital Signature Schemes in Blockchains

- one of the two most ubiquitous cryptographic primitives used in blockchain protocols (along with cryptographic hash functions)

Application #1: allows a user of a blockchain to authorize a transaction (e.g., making a payment).

- fundamental to the vision of shared computer in the sky

Digital Signature Schemes in Blockchains

- one of the two most ubiquitous cryptographic primitives used in blockchain protocols (along with cryptographic hash functions)

Application #1: allows a user of a blockchain to authorize a transaction (e.g., making a payment).

- fundamental to the vision of shared computer in the sky

Application #2: under the hood, allows validators of a blockchain protocol to sign their messages.

- used in most blockchain protocols for this purpose
  - with Bitcoin a notable exception

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   – in some cases, may generate r itself (e.g., ssh-keygen)

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   – in some cases, may generate r itself (e.g., ssh-keygen)
2. *Signing algorithm:* maps message + sk → signature.
   – signature depends on both sk and the message being signed

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   - in some cases, may generate r itself (e.g., ssh-keygen)
2. *Signing algorithm:* maps message + sk → signature.
   - signature depends on both sk and the message being signed
3. *Verification algorithm:* maps msg + sig + pk → "yes"/"no".
   - anyone who knows pk can verify correctness of an alleged signature

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   - in some cases, may generate r itself (e.g., ssh-keygen)
2. *Signing algorithm:* maps message + sk → signature.
   - signature depends on both sk and the message being signed
3. *Verification algorithm:* maps msg + sig + pk → "yes"/"no".
   - anyone who knows pk can verify correctness of an alleged signature

Ideal signature scheme: can't produce valid signatures (that you haven't already seen) unless you know the private key sk.

Defining Security for a DSS

Ideal signature scheme: can't produce valid signatures (that you haven't already seen) unless you know the private key sk.

- note: not literally true (e.g., could reverse engineer sk by brute force)

Defining Security for a DSS

Ideal signature scheme: can’t produce valid signatures (that you haven’t already seen) unless you know the private key sk.

- note: not literally true (e.g., could reverse engineer sk by brute force)

For a formal security guarantee: need to assume…

Defining Security for a DSS

Ideal signature scheme: can’t produce valid signatures (that you haven’t already seen) unless you know the private key sk.

- note: not literally true (e.g., could reverse engineer sk by brute force)

For a formal security guarantee: need to assume…

- attacker is computationally bounded (polynomial-time)

Defining Security for a DSS

Ideal signature scheme: can’t produce valid signatures (that you haven’t already seen) unless you know the private key sk.

- note: not literally true (e.g., could reverse engineer sk by brute force)

For a formal security guarantee: need to assume…

- attacker is computationally bounded (polynomial-time)
- secret key length is sufficiently long (so brute force infeasible)

Defining Security for a DSS

Ideal signature scheme: can’t produce valid signatures (that you haven’t already seen) unless you know the private key sk.

– note: not literally true (e.g., could reverse engineer sk by brute force)

For a formal security guarantee: need to assume…

- attacker is computationally bounded (polynomial-time)
- secret key length is sufficiently long (so brute force infeasible)
- no way to forge signatures much faster than brute-forcing sk
  – ideally, related to “standard” hardness assumption (like discrete log)

Defining Security for a DSS

Ideal signature scheme: can’t produce valid signatures (that you haven’t already seen) unless you know the private key sk.

– note: not literally true (e.g., could reverse engineer sk by brute force)

For a formal security guarantee: need to assume…

- attacker is computationally bounded (polynomial-time)
- secret key length is sufficiently long (so brute force infeasible)
- no way to forge signatures much faster than brute-forcing sk
  – ideally, related to “standard” hardness assumption (like discrete log)
- non-zero (but negligible) chance an attacker gets lucky

Defining Security for a DSS

For a formal security guarantee: need to assume…

- attacker is computationally bounded (polynomial-time)
- secret key length is sufficiently long (so brute force infeasible)
- no way to forge signatures much faster than brute-forcing sk
  - ideally, related to "standard" hardness assumption (like discrete log)
- non-zero (but negligible) chance an attacker gets lucky

(Semi-)formal DSS security statement: under suitable complexity assumptions, no randomized poly-time (in key length) algorithm with access to a bunch of signed messages can produce a valid signature for an unseen message with non-negligible probability.

What Signatures Can and Can’t Do

Issues:

- Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)
- Byzantine leader could propose different chains to different validators, all supported by acks from Byzantine validators. (➔ consistency violation)
- Byzantine validators could lie about messages received from other validators.

What Signatures Can and Can’t Do

Issues:

- Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)
- Byzantine leader could propose different chains to different validators, all supported by acks from Byzantine validators. (➔ consistency violation)
- Byzantine validators could lie about messages received from other validators.

Good news: signatures ➔ don’t need to worry about issue #3 (Byzantine validators can’t lie about messages sent by others).

What Signatures Can and Can't Do

Issues:

- Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)
- Byzantine leader could propose different chains to different validators, all supported by acks from Byzantine validators. (➔ consistency violation)
- Byzantine validators could lie about messages received from other validators.

Good news: signatures ➔ don't need to worry about issue #3 (Byzantine validators can't lie about messages sent by others).

Bad news: even with signatures, SMR strictly harder with Byzantine faults than with crash faults.

Recap: The Partially Synchronous Model

- shared global clock (timesteps=0,1,2,…)
- known upper bound Δ on message delays in normal conditions

Recap: The Partially Synchronous Model

- shared global clock (timesteps=0,1,2,…)
- known upper bound Δ on message delays in normal conditions
- unknown transition time GST (“global stabilization time”) from asynchrony to synchrony (i.e., end of attack/outage)
  - protocol must work no matter what GST is

Recap: The Partially Synchronous Model

- shared global clock (timesteps=0,1,2,…)
- known upper bound Δ on message delays in normal conditions
- unknown transition time GST (“global stabilization time”) from asynchrony to synchrony (i.e., end of attack/outage)
  - protocol must work no matter what GST is

Recall goals:

- consistency, always (even pre-GST/“under attack”)

Recap: The Partially Synchronous Model

- shared global clock (timesteps=0,1,2,…)
- known upper bound Δ on message delays in normal conditions
- unknown transition time GST ("global stabilization time") from asynchrony to synchrony (i.e., end of attack/outage)
  - protocol must work no matter what GST is

Recall goals:

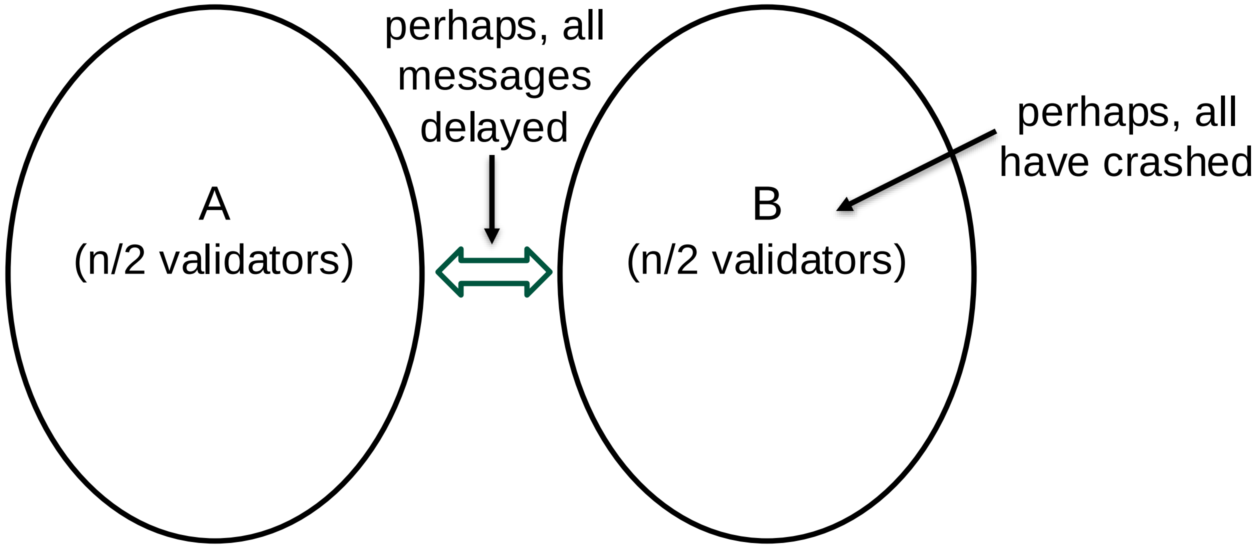
- consistency, always (even pre-GST/"under attack")
- liveness soon after GST (once "normal conditions" resume)
  - FLP ➔ need to give up one of consistency, liveness before GST

Recap: Partial Synchrony + Crash Faults

Fact: crash faults + partial synchrony ➔ security threshold < 50%.

Suppose: validators in A don't hear from any validators in B for a long time.

- should they finalize any new txs?



Catch-22:

- if validators in A wait ➔ possible liveness violation
  - if post-GST and all validators in B have crashed (will wait forever)
- if validators in A proceed ➔ possible consistency violation
  - if pre-GST and all messages A ⇔ B have been delayed

What Is Possible with Byzantine Faults?

Fact: Byzantine faults + partial synch ➔ security threshold < 33%.

- i.e., no hope unless > two-thirds of validators are non-faulty

Intuition:

What Is Possible with Byzantine Faults?

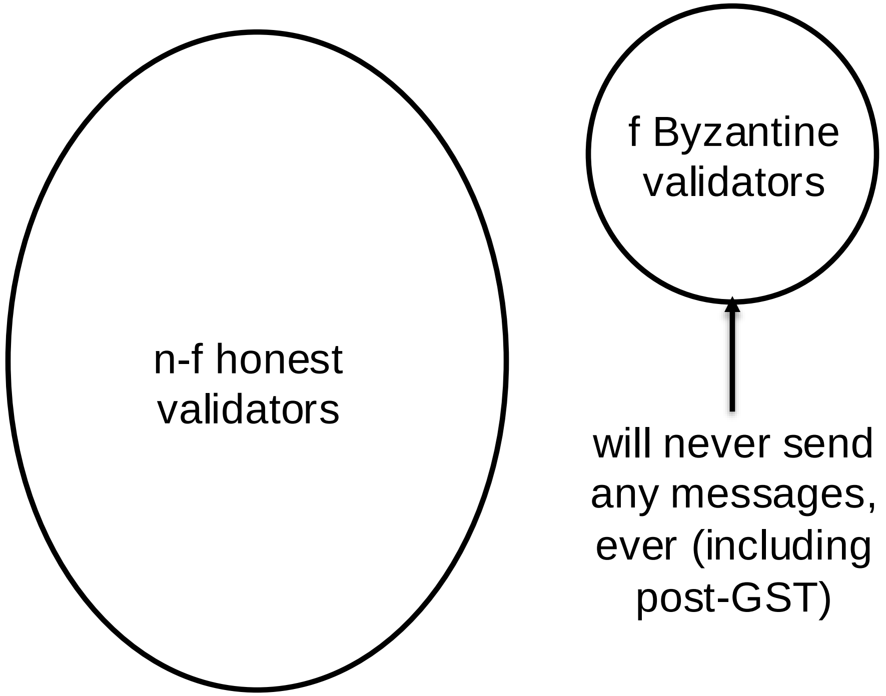
Fact: Byzantine faults + partial synch ➔ security threshold < 33%.

– i.e., no hope unless > two-thirds of validators are non-faulty

Intuition: Suppose want to tolerate up to f Byzantine faults.

1. liveness ➔ protocol must eventually finalize new transactions even if have heard from only n-f validators.
   – other f might well be Byzantine, could otherwise stall protocol forever

Post-GST Crashes or Pre-GST Delays?



Scenario #1

What Is Possible with Byzantine Faults?

Fact: Byzantine faults + partial synch ➔ security threshold < 33%.

– i.e., no hope unless > two-thirds of validators are non-faulty

Intuition: Suppose want to tolerate up to f Byzantine faults.

1. liveness ➔ protocol must eventually finalize new transactions even if have heard from only n-f validators.
   – other f might well be Byzantine, could otherwise stall protocol forever

What Is Possible with Byzantine Faults?

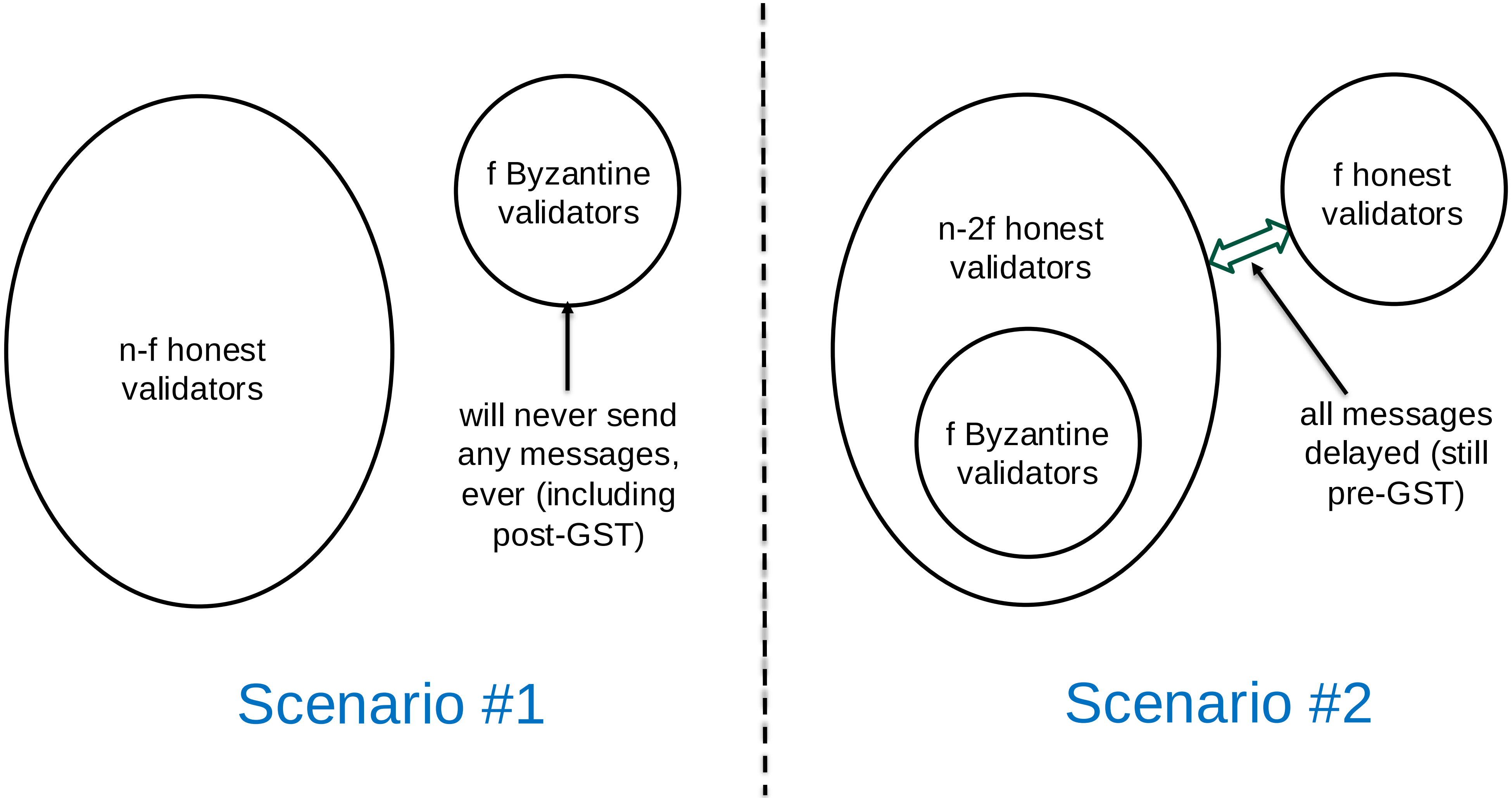
Fact: Byzantine faults + partial synch ➔ security threshold < 33%.

- i.e., no hope unless > two-thirds of validators are non-faulty

Intuition: Suppose want to tolerate up to f Byzantine faults.

1. liveness ➔ protocol must eventually finalize new transactions even if have heard from only n-f validators.
   - other f might well be Byzantine, could otherwise stall protocol forever
2. ambiguity between crashes and long msg delays ➔ might well be that f of the n-f contributing validators are Byzantine

Post-GST Crashes or Pre-GST Delays?



What Is Possible with Byzantine Faults?

Fact: Byzantine faults + partial synch ➔ security threshold < 33%.

- i.e., no hope unless > two-thirds of validators are non-faulty

Intuition: Suppose want to tolerate up to f Byzantine faults.

1. liveness ➔ protocol must eventually finalize new transactions even if have heard from only n-f validators.
2. ambiguity between crashes and long msg delays ➔ might well be that f of the n-f contributing validators are Byzantine

What Is Possible with Byzantine Faults?

Fact: Byzantine faults + partial synch ➔ security threshold < 33%.

- i.e., no hope unless > two-thirds of validators are non-faulty

Intuition: Suppose want to tolerate up to f Byzantine faults.

1. liveness ➔ protocol must eventually finalize new transactions even if have heard from only n-f validators.
2. ambiguity between crashes and long msg delays ➔ might well be that f of the n-f contributing validators are Byzantine
3. to avoid getting tricked, need strict majority of these n-f validators to be honest: $\underbrace{(n-f)-f}_{\text{honest}} > \underbrace{f}_{\text{Byzantine}}$

What Is Possible with Byzantine Faults?

Fact: Byzantine faults + partial synch ➔ security threshold < 33%.

- i.e., no hope unless > two-thirds of validators are non-faulty

Intuition: Suppose want to tolerate up to f Byzantine faults.

1. liveness ➔ protocol must eventually finalize new transactions even if have heard from only n-f validators.
2. ambiguity between crashes and long msg delays ➔ might well be that f of the n-f contributing validators are Byzantine
3. to avoid getting tricked, need strict majority of these n-f validators to be honest: $(n-f)-f > f$, i.e., $f < n/3$

(n-f)-f: honest; f: Byzantine

Toward Tendermint

Starting point: Protocol C (≈ Paxos/Raft).

Toward Tendermint

Starting point: Protocol C (≈ Paxos/Raft).

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Toward Tendermint

Starting point: Protocol C (≈ Paxos/Raft).

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Issue #2: Byzantine leader could propose different chains to different validators, all supported by acks from Byzantine validators. (➔ consistency violation)

Toward Tendermint

Starting point: Protocol C (≈ Paxos/Raft).

Issue #1: Byzantine leader could ignore read quorum requirement and make a proposal anyway. (➔ consistency violation)

Issue #2: Byzantine leader could propose different chains to different validators, all supported by acks from Byzantine validators. (➔ consistency violation)

Issue #3: Byzantine validators could lie about messages received from other validators.

Key Ideas in Tendermint

Idea #1: every validator signs every message it sends.

- assume all validators know each others public keys (+ IDs + IP addrs)
- called a “public key infrastructure (PKI)” assumption

Key Ideas in Tendermint

Idea #1: every validator signs every message it sends.

- assume all validators know each others public keys (+ IDs + IP addrs)
- called a "public key infrastructure (PKI)" assumption

Recall: in Protocol C, crucial that every write quorum (size > n/2) intersects every subsequent read quorum (size > n/2).

- reason: once a leader is in a position to make a proposal, it must be up-to-date on all txs already finalized by some non-faulty validator

Key Ideas in Tendermint

Idea #1: every validator signs every message it sends.

- assume all validators know each others public keys (+ IDs + IP addrs)
- called a "public key infrastructure (PKI)" assumption

Recall: in Protocol C, crucial that every write quorum (size > n/2) intersects every subsequent read quorum (size > n/2).

- reason: once a leader is in a position to make a proposal, it must be up-to-date on all txs already finalized by some non-faulty validator

Issue: a Byzantine validator in a read quorum could deliberately submit an out-of-date chain (ignoring its past write quorums).

Key Ideas in Tendermint (con'd)

Recall: in Protocol C, crucial that every write quorum (size > n/2) intersects every subsequent read quorum (size > n/2).

- reason: once a leader is in a position to make a proposal, it must be up-to-date on all txs already finalized by some non-faulty validator

Issue: a Byzantine validator in a read quorum could deliberately submit an out-of-date chain (ignoring its past write quorums).

Key Ideas in Tendermint (con’d)

Recall: in Protocol C, crucial that every write quorum (size > n/2) intersects every subsequent read quorum (size > n/2).

- reason: once a leader is in a position to make a proposal, it must be up-to-date on all txs already finalized by some non-faulty validator

Issue: a Byzantine validator in a read quorum could deliberately submit an out-of-date chain (ignoring its past write quorums).

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

- can count on this non-faulty validator to keep leader up-to-date

Key Ideas in Tendermint (con’d)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

- can count on this non-faulty validator to keep leader up-to-date

Idea #2:

Key Ideas in Tendermint (con'd)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

– can count on this non-faulty validator to keep leader up-to-date

Idea #2: (i) assume < n/3 validators are Byzantine. [necessary]

Key Ideas in Tendermint (con'd)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

– can count on this non-faulty validator to keep leader up-to-date

Idea #2: (i) assume < n/3 validators are Byzantine. [necessary]

(ii) increase all quorum sizes to > 2n/3 validators

Key Ideas in Tendermint (con'd)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

- can count on this non-faulty validator to keep leader up-to-date

Idea #2: (i) assume < n/3 validators are Byzantine. [necessary]

(ii) increase all quorum sizes to > 2n/3 validators

- note: given (i), (ii) does not immediately threaten liveness

Key Ideas in Tendermint (con'd)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

– can count on this non-faulty validator to keep leader up-to-date

Idea #2: (i) assume < n/3 validators are Byzantine. [necessary]

(ii) increase all quorum sizes to > 2n/3 validators

– note: given (i), (ii) does not immediately threaten liveness

- updated quorum intersection property:

Key Ideas in Tendermint (con'd)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

- can count on this non-faulty validator to keep leader up-to-date

Idea #2: (i) assume < n/3 validators are Byzantine. [necessary]

(ii) increase all quorum sizes to > 2n/3 validators

- note: given (i), (ii) does not immediately threaten liveness

- updated quorum intersection property: if S, T are quorums ➔ |S|, |T| > 2n/3

Key Ideas in Tendermint (con'd)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

– can count on this non-faulty validator to keep leader up-to-date

Idea #2: (i) assume < n/3 validators are Byzantine. [necessary]

(ii) increase all quorum sizes to > 2n/3 validators

– note: given (i), (ii) does not immediately threaten liveness

- updated quorum intersection property: if S, T are quorums ➔ |S|, |T| > 2n/3 ➔ S, T overlap in > n – n/3 – n/3 = n/3 validators

Key Ideas in Tendermint (con'd)

Fix: ensure that every read quorum, write quorum overlap in at least one *non-faulty* validator.

– can count on this non-faulty validator to keep leader up-to-date

Idea #2: (i) assume < n/3 validators are Byzantine. [necessary]

(ii) increase all quorum sizes to > 2n/3 validators

– note: given (i), (ii) does not immediately threaten liveness

- updated quorum intersection property: if S, T are quorums ➔ |S|, |T| > 2n/3 ➔ S, T overlap in > n – n/3 – n/3 = n/3 validators ➔ S, T overlap in at least one non-faulty validator

Key Ideas in Tendermint (con'd)

Idea #2: (i) assume < n/3 validators are Byzantine.

(ii) increase all quorum sizes to > 2n/3 validators

- consequence: any two quorums have non-faulty validator in common

Key Ideas in Tendermint (con’d)

Idea #2: (i) assume < n/3 validators are Byzantine.

(ii) increase all quorum sizes to > 2n/3 validators

– consequence: any two quorums have non-faulty validator in common

Bonus: can’t have write quorums for two different chains in the same view (despite equivocating leader, Byzantine acks).

- will ensure that simultaneous updates must be consistent

Key Ideas in Tendermint (con’d)

Idea #2: (i) assume < n/3 validators are Byzantine.

(ii) increase all quorum sizes to > 2n/3 validators

- consequence: any two quorums have non-faulty validator in common

Bonus: can’t have write quorums for two different chains in the same view (despite equivocating leader, Byzantine acks).

- will ensure that simultaneous updates must be consistent
- reason: non-faulty validators will ack only one proposal per view
  - two write quorums ➔ have a non-faulty validator in common ➔ validator only acked one proposal ➔ both WQs support same proposal

Key Ideas in Tendermint (con’d)

Idea #3: can’t trust leader to assemble a read quorum ➔ each validator assembles one itself before acking a proposal.

Key Ideas in Tendermint (con’d)

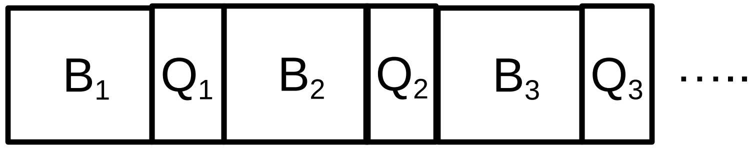
Idea #3: can’t trust leader to assemble a read quorum ➔ each validator assembles one itself before acking a proposal.

- quorum certificate (QC): > 2n/3 validators attesting that a proposal by leader is up-to-date (as far as they can tell)

Key Ideas in Tendermint (con’d)

Idea #3: can’t trust leader to assemble a read quorum ➔ each validator assembles one itself before acking a proposal.

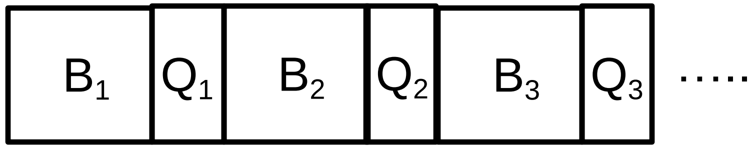
- quorum certificate (QC): > 2n/3 validators attesting that a proposal by leader is up-to-date (as far as they can tell)
  - QCs included in blockchain as metadata



Key Ideas in Tendermint (con’d)

Idea #3: can’t trust leader to assemble a read quorum ➔ each validator assembles one itself before acking a proposal.

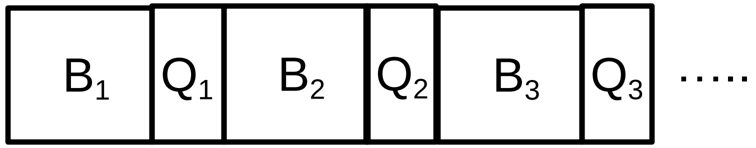
- quorum certificate (QC): > 2n/3 validators attesting that a proposal by leader is up-to-date (as far as they can tell)
  - QCs included in blockchain as metadata
  - adds extra round to each view



Key Ideas in Tendermint (con’d)

Idea #3: can’t trust leader to assemble a read quorum ➔ each validator assembles one itself before acking a proposal.

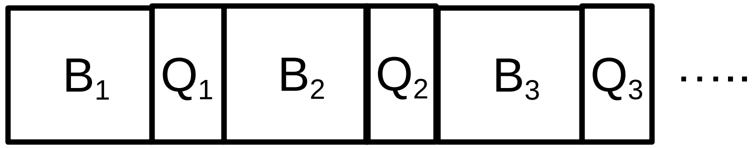
- quorum certificate (QC): > 2n/3 validators attesting that a proposal by leader is up-to-date (as far as they can tell)
  - QCs included in blockchain as metadata
  - adds extra round to each view
  - note: QCs don’t even make sense without idea #1 (signatures)



Key Ideas in Tendermint (con’d)

Idea #3: can’t trust leader to assemble a read quorum ➔ each validator assembles one itself before acking a proposal.

- quorum certificate (QC): > 2n/3 validators attesting that a proposal by leader is up-to-date (as far as they can tell)
  - QCs included in blockchain as metadata
  - adds extra round to each view
  - note: QCs don’t even make sense without idea #1 (signatures)

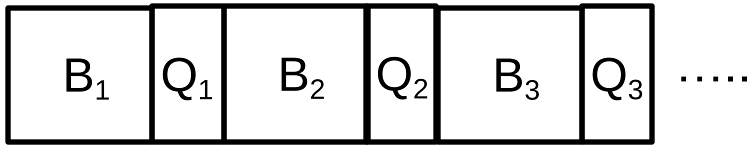


Worry: Byzantine validators will manipulate QC formation.

Key Ideas in Tendermint (con’d)

Idea #3: can’t trust leader to assemble a read quorum ➔ each validator assembles one itself before acking a proposal.

- quorum certificate (QC): > 2n/3 validators attesting that a proposal by leader is up-to-date (as far as they can tell)
  - QCs included in blockchain as metadata
  - adds extra round to each view
  - note: QCs don’t even make sense without idea #1 (signatures)



Worry: Byzantine validators will manipulate QC formation.

  - good news: idea #2 ➔ impossible to have QCs for two different proposals in the same view (effectively, equivocation not possible)