Lecture #23: Proof-of-Stake Blockchain Protocols

COMS 4995-001: The Science of Blockchains

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

Tim Roughgarden

Goals for Lecture #23

1. Weighted round-robin.
   – quick and dirty approach to stake-proportional leader selection
2. Degrees of permissionlessness.
   – sense in which PoS protocols are “more permissioned” than PoW
3. A proof-of-stake Tendermint.
   – consistent and live in partial synchrony under appropriate assumptions
4. Slashing.
   – programmatically fight back against a 51%-type attack

Weighted Round Robin

Given: list $(pk_1,q_1),\ldots, (pk_n,q_n)$ of active validators + stake amounts.

Goal: sample pk from $\{pk_1,\ldots,pk_n\}$ with probability proportional to q_i's.

Solution:

Weighted Round Robin

Given: list $(pk_1,q_1),\ldots, (pk_n,q_n)$ of active validators + stake amounts.

Goal: sample pk from $\{pk_1,\ldots,pk_n\}$ with probability proportional to q_i's.

Solution: use epoch of length N views each (N large).

- list of active validators + their stakes changes only at epoch boundaries
- each epoch: use proportionally representative sequence of leaders

Weighted Round Robin

Given: list $(pk_1,q_1),\ldots,(pk_n,q_n)$ of active validators + stake amounts.

Goal: sample pk from $\{pk_1,\ldots,pk_n\}$ with probability proportional to q_i's.

Solution: use epoch of length N views each (N large).

- list of active validators + their stakes changes only at epoch boundaries
- each epoch: use proportionally representative sequence of leaders
- ex: {(A,2),(B,1),(C,2)} ➔ use leader sequence AABCCAABCCAABCC…

Weighted Round Robin

Given: list $(pk_1,q_1),\ldots, (pk_n,q_n)$ of active validators + stake amounts.

Goal: sample pk from $\{pk_1,\ldots,pk_n\}$ with probability proportional to q_i's.

Solution: use epoch of length N views each (N large).

- list of active validators + their stakes changes only at epoch boundaries
- each epoch: use proportionally representative sequence of leaders
- ex: {(A,2),(B,1),(C,2)} ➔ use leader sequence AABCCAABCCAABCC…

Good news: relatively simple, no fancy cryptography.

Weighted Round Robin

Given: list $(pk_1,q_1),\ldots, (pk_n,q_n)$ of active validators + stake amounts.

Goal: sample pk from $\{pk_1,\ldots,pk_n\}$ with probability proportional to q_i's.

Solution: use epoch of length N views each (N large).

- list of active validators + their stakes changes only at epoch boundaries
- each epoch: use proportionally representative sequence of leaders
- ex: {(A,2),(B,1),(C,2)} ➔ use leader sequence AABCCAABCCAABCC…

Good news: relatively simple, no fancy cryptography.

Bad news: leaders of future views known well in advance.

Weighted Round Robin

Given: list $(pk_1,q_1),\ldots, (pk_n,q_n)$ of active validators + stake amounts.

Goal: sample pk from $\{pk_1,\ldots,pk_n\}$ with probability proportional to q_i's.

Solution: use epoch of length N views each (N large).

Good news: relatively simple, no fancy cryptography.

Bad news: leaders of future views known well in advance.

- ➔ risk of bribery, coercion, DoS attacks
- also has its benefits (e.g., for tx dissemination)

Degrees of Permissionlessness

increasing permissionlessness (and difficulty)

Degrees of Permissionlessness

Idea: parameterize "how permissionless" a protocol is according to its knowledge of and assumptions about its validators.

increasing permissionlessness (and difficulty)

Degrees of Permissionlessness

Idea: parameterize "how permissionless" a protocol is according to its knowledge of and assumptions about its validators.

- cf., parameterizing by:
  - the number of faulty validators
  - the type of fault (crash vs. Byzantine, etc.)
  - the reliability of the network (sync vs. partial sync vs. async)

increasing permissionlessness (and difficulty)

Degrees of Permissionlessness

- fully permissionless (FP): protocol knows nothing about its validators (e.g., Nakamoto consensus)

increasing permissionlessness (and difficulty)

Degrees of Permissionlessness

- fully permissionless (FP): protocol knows nothing about its validators (e.g., Nakamoto consensus)
- dynamically available (DA): protocol knows (evolving) list of pks that represent the possibly active validators
  - allows some of these pks to be not actively validating (i.e., offline)

increasing permissionlessness (and difficulty)

Degrees of Permissionlessness

- fully permissionless (FP): protocol knows nothing about its validators (e.g., Nakamoto consensus)
- dynamically available (DA): protocol knows (evolving) list of pks that represent the possibly active validators
  - allows some of these pks to be not actively validating (i.e., offline)
- quasi-permissionless (QP): protocol can also assume that all (non-faulty) list members are actively participating
  - i.e., no longer allow an "honest but offline validator"

increasing permissionlessness (and difficulty)

Degrees of Permissionlessness

- fully permissionless (FP): protocol knows nothing about its validators (e.g., Nakamoto consensus)
- dynamically available (DA): protocol knows (evolving) list of pks that represent the possibly active validators
  - allows some of these pks to be not actively validating (i.e., offline)
- quasi-permissionless (QP): protocol can also assume that all (non-faulty) list members are actively participating
  - i.e., no longer allow an "honest but offline validator"
- permissioned (Perm): validator set fixed forever when protocol is deployed (e.g., Tendermint)

increasing permissionlessness (and difficulty)

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)
- $<$ 50% Byzantine stake (necessary with longest-chain consensus)

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)
- $<$ 50% Byzantine stake (necessary with longest-chain consensus)
- dynamically available setting (slightly stronger than FP setting)

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)
- < 50% Byzantine stake (necessary with longest-chain consensus)
- dynamically available setting (slightly stronger than FP setting)
  - also: no difficulty adjustment algorithm required (stake directly observable)

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)
- $<$ 50% Byzantine stake (necessary with longest-chain consensus)
- dynamically available setting (slightly stronger than FP setting)

Bad news:

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)
- $< 50\%$ Byzantine stake (necessary with longest-chain consensus)
- dynamically available setting (slightly stronger than FP setting)

Bad news: loses consistency in partial synchrony (network partitions)

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)
- < 50% Byzantine stake (necessary with longest-chain consensus)
- dynamically available setting (slightly stronger than FP setting)

Bad news: loses consistency in partial synchrony (network partitions)

- relatively large latency (due to security parameter k)

Pairing PoS with a Consensus Protocol

Fact: possible to pair proof-of-stake sybil-resistance with longest-chain consensus (see e.g. Cardano).

Good news: consistent and live provided:

- synchronous network (necessary with longest-chain consensus)
- < 50% Byzantine stake (necessary with longest-chain consensus)
- dynamically available setting (slightly stronger than FP setting)

Bad news: loses consistency in partial synchrony (network partitions)

- relatively large latency (due to security parameter k)
- much more complex than Nakamoto consensus (see YT videos)
  – e.g., leaders can now equivocate, make multiple proposals per view

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony.

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

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony. (i.e., same as Tendermint, but permissionless)

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony. (i.e., same as Tendermint, but permissionless)

From last week: impossible in the dynamically available setting.

- even for proof-of-stake protocols

Limitations of Proof-of-Work

1. no PoW protocol is consistent in partial synchrony
2. even in synchrony, no POW protocol guarantees (deterministic) consistency + liveness

Intuition for (1): catch-22 a la “CAP principle” argument. If validator hears no messages for a long time, can’t distinguish between:

- (i) in synchrony, other validators turned off their machines
- (ii) in partial synchrony + pre-GST, all messages delayed

Should the validator ever finalize any additional txs?

- yes ➔ might be in scenario (ii), cause a consistency violation
- no ➔ might be in scenario (i), liveness violation (in synchrony)

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony. (i.e., same as Tendermint, but permissionless)

From last week: impossible in the dynamically available setting.

- even for proof-of-stake protocols
- issue: ambiguity between Byzantine and honest offline validators

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony. (i.e., same as Tendermint, but permissionless)

From last week: impossible in the dynamically available setting.

- even for proof-of-stake protocols
- issue: ambiguity between Byzantine and honest offline validators
- recall original challenge: how big should a quorum be?

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony. (i.e., same as Tendermint, but permissionless)

From last week: impossible in the dynamically available setting.

- even for proof-of-stake protocols
- issue: ambiguity between Byzantine and honest offline validators
- recall original challenge: how big should a quorum be?

Hope: possible in the quasi-permissionless setting.

- recall: offline validators are considered faulty in the QP setting

A Proof-of-Stake Version of Tendermint

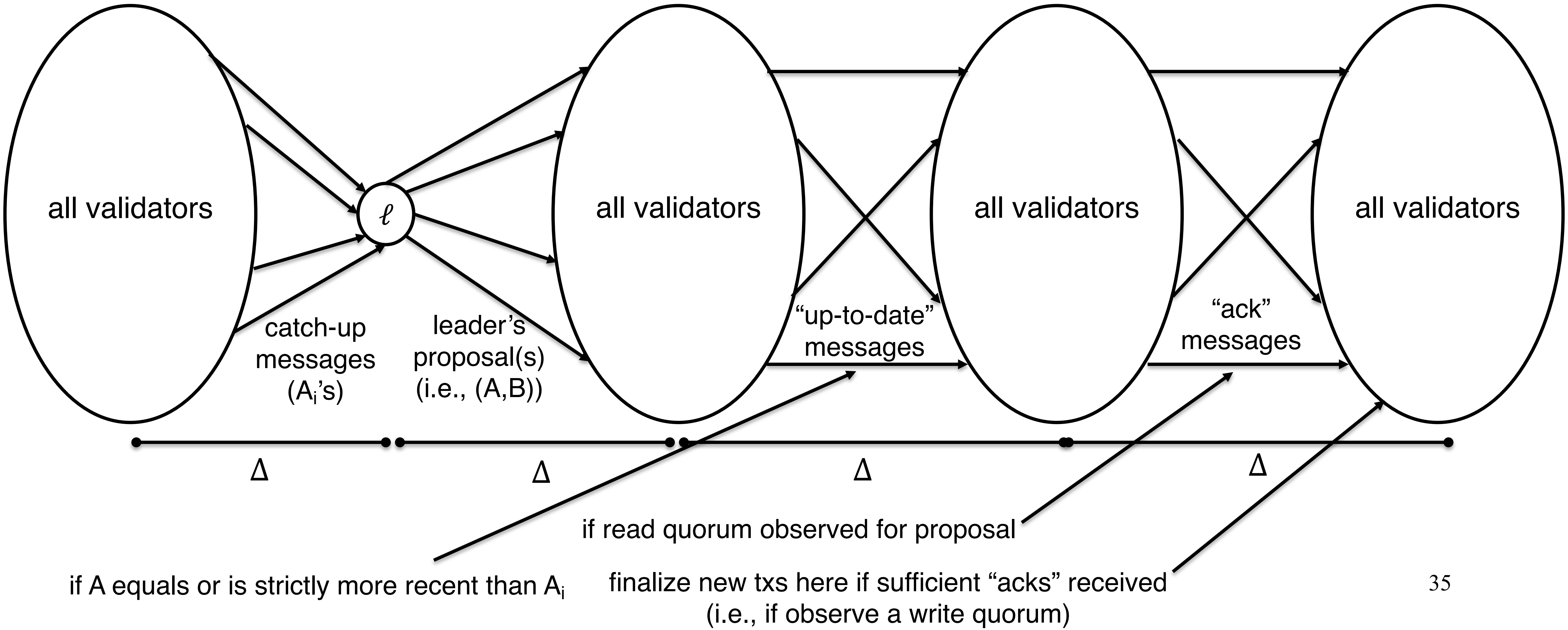
Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony, in the quasi-permissionless setting.

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony, in the quasi-permissionless setting.

Simplest approach: modify Tendermint so that:

Tendermint: Picture of One View



Tendermint: Pseudocode

- at time $4\Delta \cdot v$:
  - each validator i sends its current chain A_i to v's leader ℓ
- at time $4\Delta \cdot v + \Delta$:
  - let A = of the A_i's received, the most recently created one; let B := all not-yet-included (in A) valid txs ℓ knows about
  - ℓ sends proposal (A,B) to all other validators
- at time $4\Delta \cdot v + 2\Delta$:
  - if validator i receives a proposal (A,B) from ℓ with A = A_i or with A more recent than A_i by this time:
    - send "(A,B) is up-to-date" message to all validators
- at time $4\Delta \cdot v + 3\Delta$:
  - if validator i has heard > 2n/3 "up-to-date" msgs for (A,B) by this time (a *read quorum)*:
    - package these messages into a quorum certificate (QC), Q
    - send "ack (A,B,Q)" message to all validators and reset A_i := (A,B,Q)
- at time $4\Delta \cdot v + 4\Delta$:
  - if validator i has received > 2n/3 "ack (A,B,Q)" messages (a *write quorum)*:
    - reset C_i := (A,B,Q) (and also A_i := (A,B,Q), if necessary)

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony, in the quasi-permissionless setting.

Simplest approach: modify Tendermint so that:

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony, in the quasi-permissionless setting.

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
  - validators only allowed to join/leave at epoch boundaries

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony, in the quasi-permissionless setting.

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
  - validators only allowed to join/leave at epoch boundaries
- redefine quorum certificate = signatures by distinct public keys that collectively represent more than 2/3rds of the overall stake

A Proof-of-Stake Version of Tendermint

Goal: a proof-of-stake protocol that is consistent and (eventually) live in partial synchrony, in the quasi-permissionless setting.

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
  - validators only allowed to join/leave at epoch boundaries
- redefine quorum certificate = signatures by distinct public keys that collectively represent more than 2/3rds of the overall stake
- add logic to update validator set at epoch boundaries
  - warning: can be hard to get right (without a synchrony assumption)

A Proof-of-Stake Version of Tendermint

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
- redefine quorum certificate = signatures by distinct public keys that collectively represent more than 2/3rds of the overall stake
- add logic to update validator set at epoch boundaries

Result: assuming

A Proof-of-Stake Version of Tendermint

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
- redefine quorum certificate = signatures by distinct public keys that collectively represent more than 2/3rds of the overall stake
- add logic to update validator set at epoch boundaries

Result: assuming (i) $\leq$ 33% of stake controlled by Byzantine validators (at all times)

A Proof-of-Stake Version of Tendermint

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
- redefine quorum certificate = signatures by distinct public keys that collectively represent more than 2/3rds of the overall stake
- add logic to update validator set at epoch boundaries

Result: assuming (i) $\leq$ 33% of stake controlled by Byzantine validators (at all times) and (ii) quasi-permissionless setting

A Proof-of-Stake Version of Tendermint

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
- redefine quorum certificate = signatures by distinct public keys that collectively represent more than 2/3rds of the overall stake
- add logic to update validator set at epoch boundaries

Result: assuming (i) ≤ 33% of stake controlled by Byzantine validators (at all times) and (ii) quasi-permissionless setting ➔ proof-of-stake Tendermint is consistent + live in partial synchrony.

A Proof-of-Stake Version of Tendermint

Simplest approach: modify Tendermint so that:

- uses epoch-based weighted round-robin leader selection
- redefine quorum certificate = signatures by distinct public keys that collectively represent more than 2/3rds of the overall stake
- add logic to update validator set at epoch boundaries

Result: assuming (i) ≤ 33% of stake controlled by Byzantine validators (at all times) and (ii) quasi-permissionless setting ➔ proof-of-stake Tendermint is consistent + live in partial synchrony.

In general: can usually (not always) turn a permissioned protocol into a PoS protocol with the same guarantees in the QP setting.

Slashing

Idea: confiscate stake of validators that deviate from protocol.

- taking advantage of cool-down period + native currency

Slashing

Idea: confiscate stake of validators that deviate from protocol.

- taking advantage of cool-down period + native currency

Version #1: manual/hard fork (a.k.a. “social slashing”).

- – convince honest validators to run new version of protocol with balances of deviating validators zeroed out (note: no analog in PoW)
- – ideally used only as last resort/nuclear option

Slashing

Idea: confiscate stake of validators that deviate from protocol.

- taking advantage of cool-down period + native currency

Version #1: manual/hard fork (a.k.a. “social slashing”).

- – convince honest validators to run new version of protocol with balances of deviating validators zeroed out (note: no analog in PoW)
- – ideally used only as last resort/nuclear option

Version #2: programmatic (i.e., in-protocol) slashing.

Slashing

Idea: confiscate stake of validators that deviate from protocol.

Version #1: manual/hard fork (a.k.a. “social slashing”).

Version #2: programmatic (i.e., in-protocol) slashing.

- prerequisite #1: detectable deviations

Slashing

Idea: confiscate stake of validators that deviate from protocol.

Version #1: manual/hard fork (a.k.a. “social slashing”).

Version #2: programmatic (i.e., in-protocol) slashing.

- prerequisite #1: detectable deviations
  - example: voting for conflicting blocks in Tendermint (consistency violation ➔ > 33% of the stake must do this)

Slashing

Idea: confiscate stake of validators that deviate from protocol.

Version #1: manual/hard fork (a.k.a. "social slashing").

Version #2: programmatic (i.e., in-protocol) slashing.

- prerequisite #1: detectable deviations
  - example: voting for conflicting blocks in Tendermint (consistency violation ➔ > 33% of the stake must do this)
  - trickier example: failing to submit block proposal or vote on time

Slashing

Idea: confiscate stake of validators that deviate from protocol.

Version #1: manual/hard fork (a.k.a. "social slashing").

Version #2: programmatic (i.e., in-protocol) slashing.

- prerequisite #1: detectable deviations
  - example: voting for conflicting blocks in Tendermint (consistency violation ➔ > 33% of the stake must do this)
  - trickier example: failing to submit block proposal or vote on time
    - see "inactivity leaks" in Ethereum

Slashing

Idea: confiscate stake of validators that deviate from protocol.

Version #1: manual/hard fork (a.k.a. "social slashing").

Version #2: programmatic (i.e., in-protocol) slashing.

- prerequisite #1: detectable deviations
  - example: voting for conflicting blocks in Tendermint (consistency violation ➔ > 33% of the stake must do this)
  - trickier example: failing to submit block proposal or vote on time
    - see "inactivity leaks" in Ethereum
    - issue: could have been caused by unreliable network or censoring by other validators, rather than Byzantine behavior

Slashing

Idea: confiscate stake of validators that deviate from protocol.

Version #1: manual/hard fork (a.k.a. “social slashing”).

Version #2: programmatic (i.e., in-protocol) slashing.

- prerequisite #1: detectable deviations
  - example: voting for conflicting blocks in Tendermint
  - trickier example: failing to submit block proposal or vote on time
- prerequisite #2: certificate of guilt must be posted on-chain
  - e.g., signatures by a validator on two conflicting blocks

Slashing

Idea: confiscate stake of validators that deviate from protocol.

Version #1: manual/hard fork (a.k.a. “social slashing”).

Version #2: programmatic (i.e., in-protocol) slashing.

- prerequisite #1: detectable deviations
  - example: voting for conflicting blocks in Tendermint
  - trickier example: failing to submit block proposal or vote on time
- prerequisite #2: certificate of guilt must be posted on-chain
  - e.g., signatures by a validator on two conflicting blocks
- design decision: how much to slash for various offenses?

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)
- pro #2: potential for in-protocol recovery from major attacks

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)
- pro #2: potential for in-protocol recovery from major attacks
- con #1: additional protocol complexity (e.g., to verify deviation)

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)
- pro #2: potential for in-protocol recovery from major attacks
- con #1: additional protocol complexity (e.g., to verify deviation)
- con #2: additional attack vectors (e.g., censoring validator votes)

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)
- pro #2: potential for in-protocol recovery from major attacks
- con #1: additional protocol complexity (e.g., to verify deviation)
- con #2: additional attack vectors (e.g., censoring validator votes)
- con #3: inadvertent slashing of honest validators

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)
- pro #2: potential for in-protocol recovery from major attacks
- con #1: additional protocol complexity (e.g., to verify deviation)
- con #2: additional attack vectors (e.g., censoring validator votes)
- con #3: inadvertent slashing of honest validators
- point of debate: rewards should already be sufficient incentive

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)
- pro #2: potential for in-protocol recovery from major attacks
- con #1: additional protocol complexity (e.g., to verify deviation)
- con #2: additional attack vectors (e.g., censoring validator votes)
- con #3: inadvertent slashing of honest validators
- point of debate: rewards should already be sufficient incentive
  - can resort to social slashing in exceptional situations
  - can always simulate in-protocol slashing, only more options

To Slash or Not to Slash?

Ongoing debate: in-protocol slashing, or social slashing only?

- pro #1: stronger incentives for correct behavior (carrot + stick)
- pro #2: potential for in-protocol recovery from major attacks
- con #1: additional protocol complexity (e.g., to verify deviation)
- con #2: additional attack vectors (e.g., censoring validator votes)
- con #3: inadvertent slashing of honest validators
- point of debate: rewards should already be sufficient incentive
  - can resort to social slashing in exceptional situations
  - can always simulate in-protocol slashing, only more options
  - counterpoint: social slashing likely either (i) too hard, won't actually happen when you need it; or (ii) too easy, subject to abuse

Proof-of-Stake: Some Advanced Topics

Issue #1: predictability of leaders in weighted round-robin.

Proof-of-Stake: Some Advanced Topics

Issue #1: predictability of leaders in weighted round-robin.

- more sophisticated solution: *verifiable random functions (VRFs)*
  - no one knows you're the leader until they see your block proposal

Proof-of-Stake: Some Advanced Topics

Issue #1: predictability of leaders in weighted round-robin.

- more sophisticated solution: *verifiable random functions (VRFs)*
  - no one knows you're the leader until they see your block proposal

Issue #2: too many validators ➔ bad performance in Tendermint.

Proof-of-Stake: Some Advanced Topics

Issue #1: predictability of leaders in weighted round-robin.

- more sophisticated solution: *verifiable random functions (VRFs)*
  - no one knows you're the leader until they see your block proposal

Issue #2: too many validators ➔ bad performance in Tendermint.

- solution #1: cap number of validators, explicitly or implicitly

Proof-of-Stake: Some Advanced Topics

Issue #1: predictability of leaders in weighted round-robin.

- more sophisticated solution: *verifiable random functions (VRFs)*
  - no one knows you're the leader until they see your block proposal

Issue #2: too many validators ➔ bad performance in Tendermint.

- solution #1: cap number of validators, explicitly or implicitly
- solution #2: combine Tendermint with some aspects of longest-chain consensus (which scales well with # of validators)

Proof-of-Stake: Some Advanced Topics

Issue #1: predictability of leaders in weighted round-robin.

- more sophisticated solution: *verifiable random functions (VRFs)*
  - no one knows you're the leader until they see your block proposal

Issue #2: too many validators ➔ bad performance in Tendermint.

- solution #1: cap number of validators, explicitly or implicitly
- solution #2: combine Tendermint with some aspects of longest-chain consensus (which scales well with # of validators)
- solution #3: use randomly sampled "committees" of validators