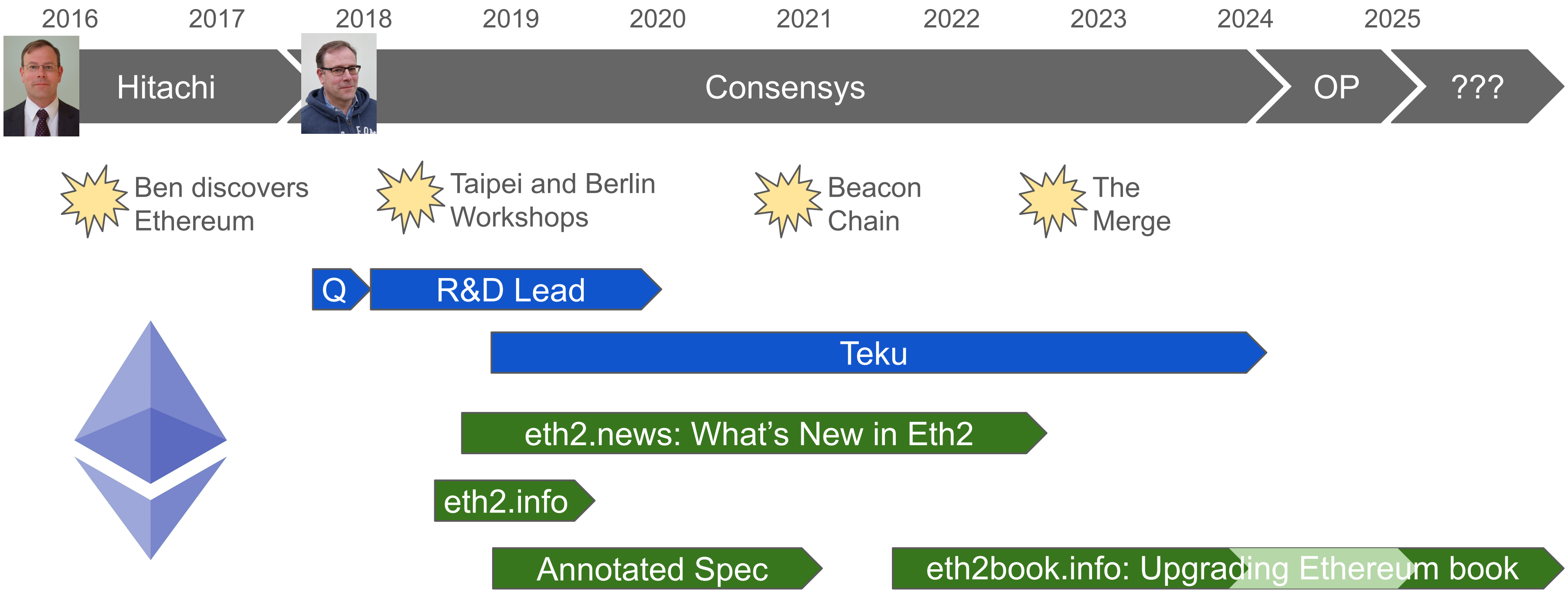
Fork Choice

Ethereum Protocol Study Group

2025-03-10

Ben’s adventures in Ethereum



Agenda

1. Introduction to Fork Choice
2. LMD GHOST
3. Casper FFG
4. Gasper
5. AMA

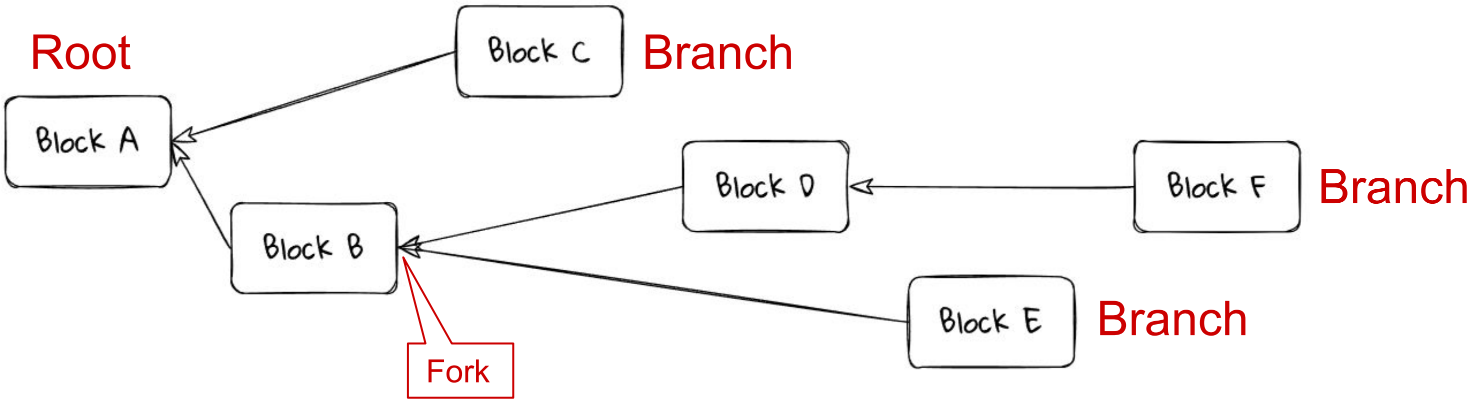
Introduction

Recap: Safety and Liveness in distributed systems

- Safety: consistency / finalisation
  - “Nothing bad ever happens”
- Liveness: availability
  - “Something good eventually happens”
- CAP Theorem
  - You can’t guarantee both! (On a real network)
- Ethereum prioritises liveness
  - LMD GHOST
- With “best efforts” safety / finality
  - Casper FFG - a “finality gadget”
- In combination: GASPER

The fork choice rule

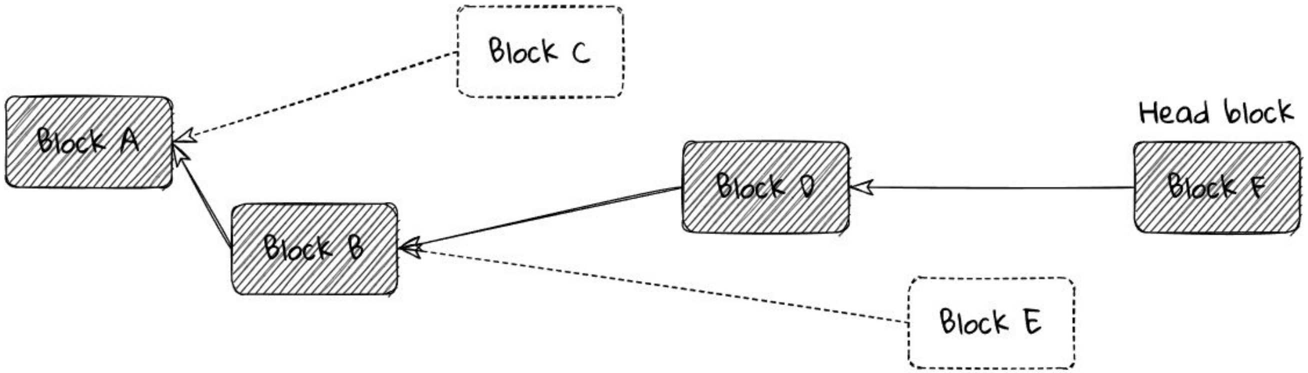
- Liveness prioritising networks are “forkful”
  - They (sometimes) build trees rather than chains.
- A symptom of the lack of safety: different nodes have different views of the network.



The fork choice rule

The “fork choice rule” is the method by which the network converges on a single branch, and therefore a single, shared history. Block tree → Blockchain.

Heuristic: build on the branch that is most likely to be built on (least likely to be orphaned) by other nodes, based on whatever information we have received.



Canonical chain

Here be Dragons!

Historical issues with Ethereum's PoS fork choice:

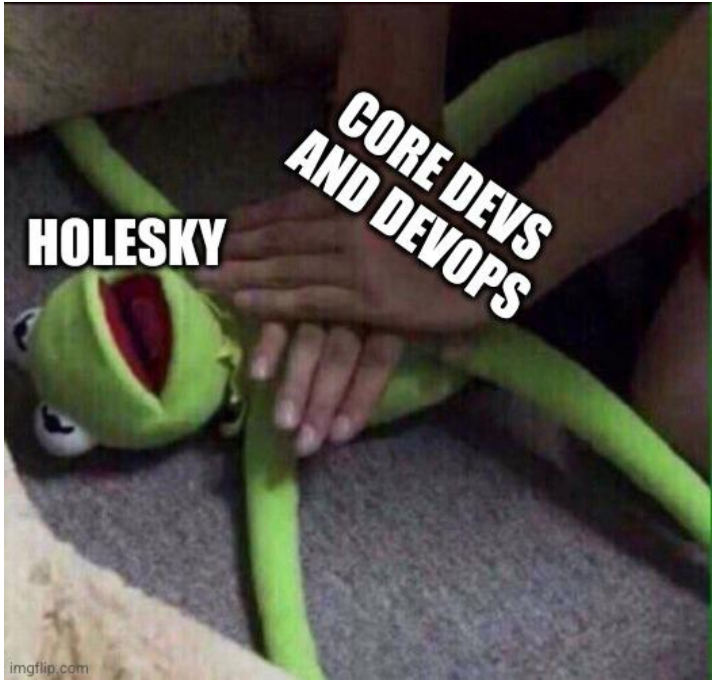
- August 2019, decoy flip-flop attack (temporary finality delay)
- September 2019, bouncing attack (indefinite finality delay)
- November 2019, finality liveness failure (unable to finalise new checkpoints)
- October 2020, balancing attack (long forks, indefinite finality delay)
- July 2021, inconsistent finalised and justified
- November 2021, non-atomic finalisation and justification
- January 2022, a new balancing attack
- May 2022, unrealised justification reorgs (proposers can induce long reorgs)
- June 2022, justification withholding attacks (ditto)

Reference

When fork choice struggles: Holesky / Pectra upgrade

“40 hours in the life of fork choice” interop-🌃Discord channel (2025-02-24 22:00)

- Faulty clients
- P2P meltdown
- Slashing protection
- Low block production
- Snap sync
- Checkpoint sync
- Optimistic sync
- Client instability (oom)
- …



h/t smartprogrammer.eth

Navigating the specs

```python
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint
```

Each validator broadcasts its view of the network via an attestation, once per epoch (32 slots, 6.4 minutes):

- Broadcast via P2P gossip
- Packed into blocks by proposers

- LMD GHOST largely depends on attestations received via gossip
  - Therefore handling is part of fork choice: *fork-choice.md*
- Casper FFG solely depends on attestations in blocks
  - Therefore handling is part of block and epoch processing: *beacon-chain.md*

The Store

Each node maintains its own view of the network in an object called [the Store](#).

```
@dataclass
class Store(object):
    time: uint64
    genesis_time: uint64
    justified_checkpoint: Checkpoint
    finalized_checkpoint: Checkpoint
  * unrealized_justified_checkpoint: Checkpoint
  * unrealized_finalized_checkpoint: Checkpoint
  * proposer_boost_root: Root
  * equivocating_indices: Set[ValidatorIndex]
    blocks: Dict[Root, BeaconBlock] = field(default_factory=dict)
    block_states: Dict[Root, BeaconState] = field(default_factory=dict)
  * block_timeliness: Dict[Root, boolean] = field(default_factory=dict)
    checkpoint_states: Dict[Checkpoint, BeaconState] = field(default_factory=dict)
    latest_messages: Dict[ValidatorIndex, LatestMessage] = field(default_factory=dict)
  * unrealized_justifications: Dict[Root, Checkpoint] = field(default_factory=dict)
```

* things added later to fix issues

Events

- The LMD GHOST fork choice is event driven.
- Handlers:
  - `on_tick()`
    - Regularly updates the current time and does some light housekeeping.
  - `on_block()`
    - Adds new blocks to the Store as the node receives them
  - `on_attestation()`
    - Updates validators' latest messages, whether received in blocks or via P2P
  - `on_attester_slashing()`
    - Ensures that slashed validators' votes are not counted, and so avoids some equivocation attacks.
- It is always ready to output a best head block via a call to `get_head()`

LMD GHOST

Naming

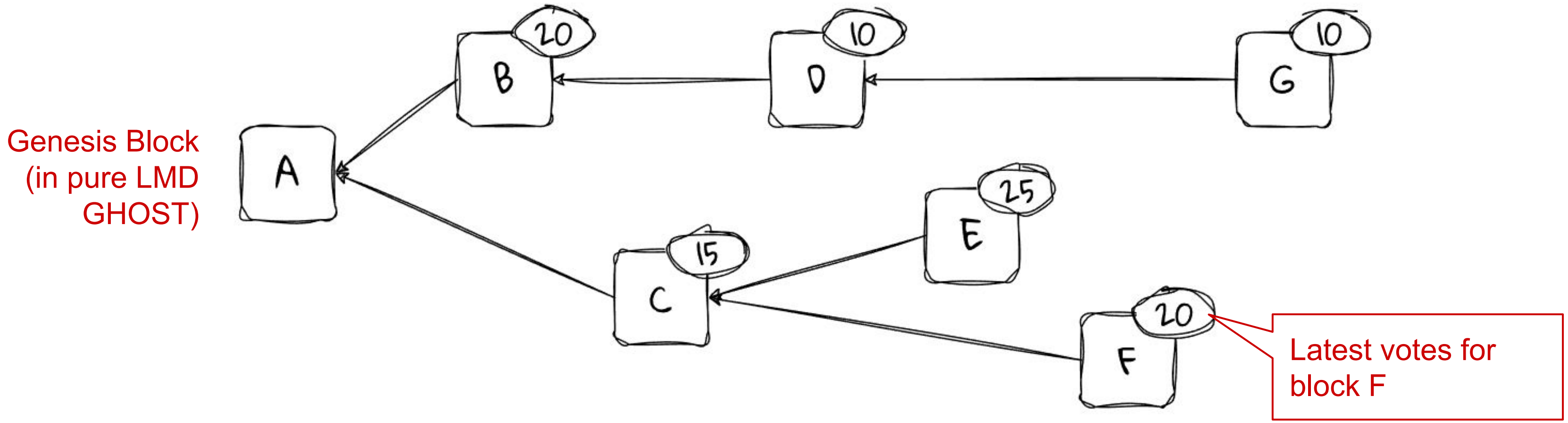
```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint
```

- LMD
  - “Message Driven”
    - Relies only on attestations
    - Validators attest to what they believe to be the best head in the current slot
  - “Latest”
    - Only the most recent attestation from each validator counts
    - IMD GHOST (immediate), FMD GHOST (fresh), RLMD GHOST (recent latest)...
- GHOST
  - “Greedy Heaviest-Observed Sub-Tree” algorithm
  - from a 2013 paper by Sompolinsky and Zohar about how to safely improve transaction throughput on Bitcoin

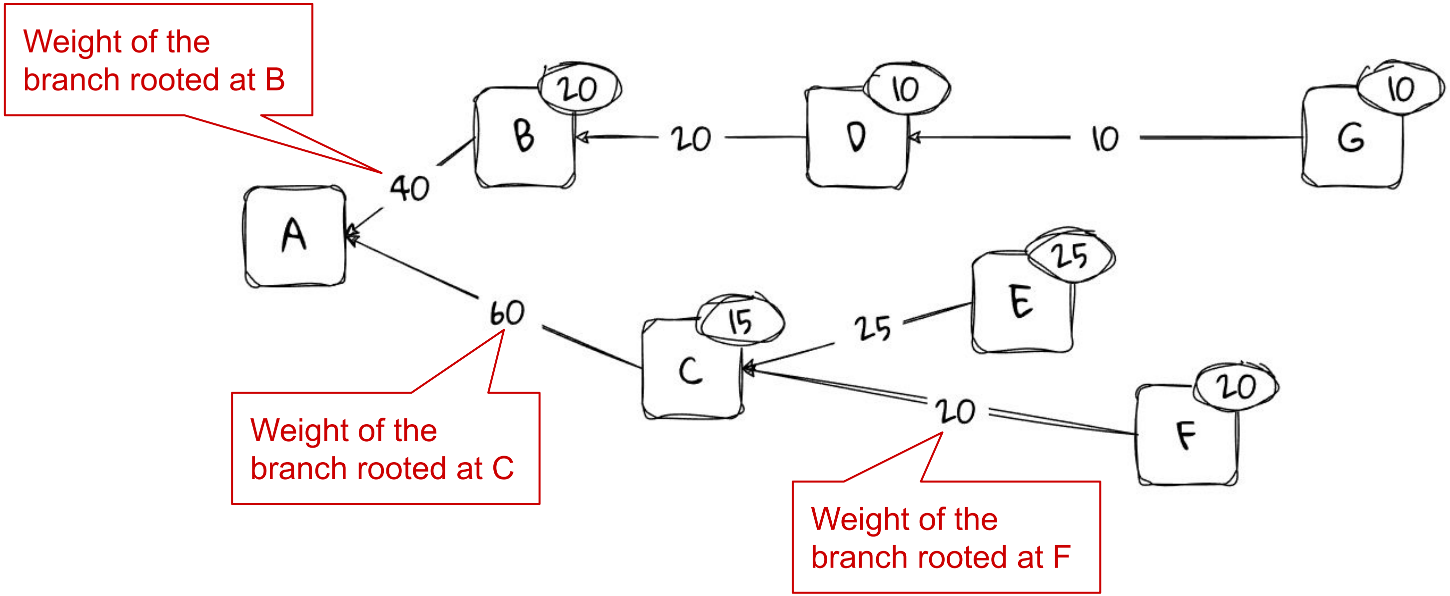
LMD GHOST Overview

- Timescale:
  - Slot-based: 12s
- Goal:
  - Used by the block proposer to decide on which branch to build its block
  - Used by attesters to choose which branch to attest to => convergence
- Heuristic:
  - Which branch is least likely to be orphaned in future?
- Based on:
  - “Weighing” the votes received in attestations
  - Note that a maximum of only 3.125% of the votes are “fresh” (same-slot)
- Properties
  - Liveness: it will always output a viable head block on which to build (proof under synchrony)
  - Safety: no useful guarantees (however, see the confirmation rule work.)

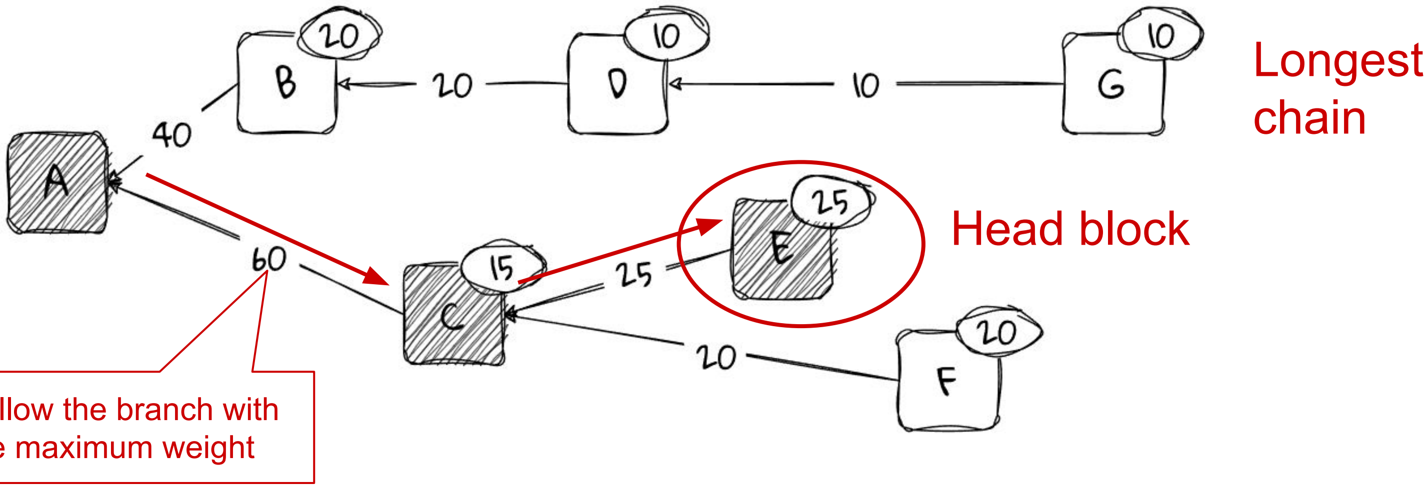
LMD GHOST fork choice



LMD GHOST fork choice



LMD GHOST fork choice



get_head()

```
def get_head(store: Store) -> Root:
    # Get filtered block tree that only includes viable branches
    blocks = get_filtered_block_tree(store)
    # Execute the LMD-GHOST fork choice
    head = store.justified_checkpoint.root
    while True:
        children = [
            root for root in blocks.keys()
            if blocks[root].parent_root == head
        ]
        if len(children) == 0:
            return head
        # Sort by latest attesting balance with ties broken lexicographically
        # Ties broken by favoring block with lexicographically higher root
        head = max(children, key=lambda root: (get_weight(store, root), root))
```

In pure LMD GHOST this would be the genesis block

(recursive function written iteratively)

Stop when we reach a branch tip

Follow the branch with the maximum weight

Weigh the branch rooted at block "`root`"

Phase 0

[fork-choice.md]

[Implementations]

`get_weight()` (of a branch rooted at block "`root`")

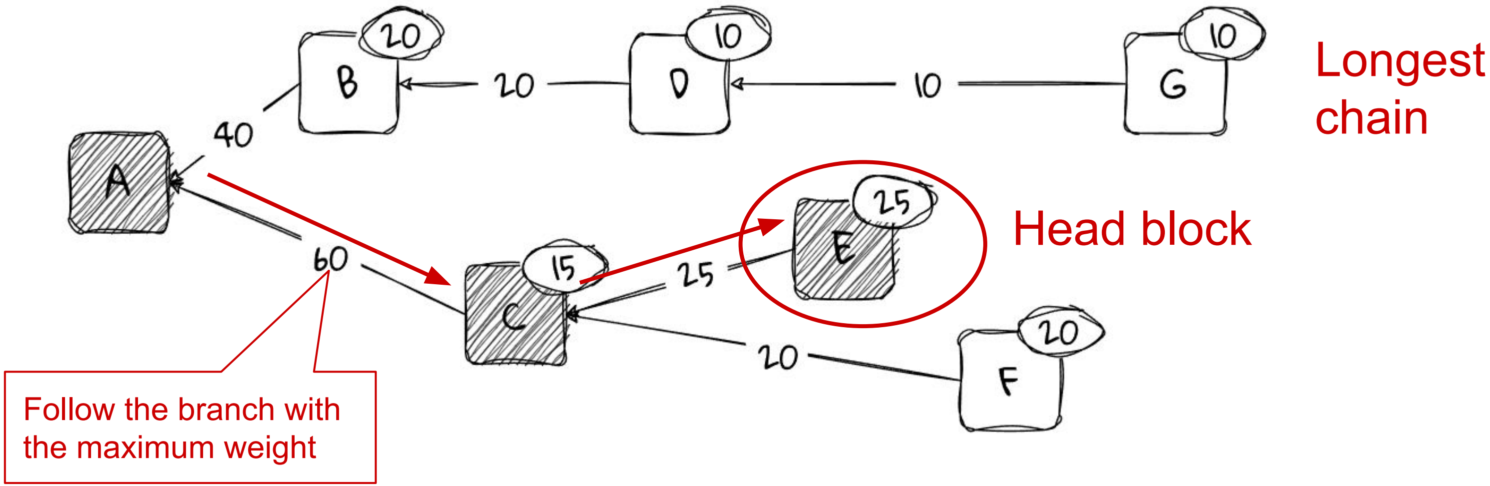
```
def get_weight(store: Store, root: Root) -> Gwei:
    state = store.checkpoint_states[store.justified_checkpoint]
    unslashed_and_active_indices = [
        i for i in get_active_validator_indices(state, get_current_epoch(state))
        if not state.validators[i].slashed
    ]
    attestation_score = Gwei(sum(
        state.validators[i].effective_balance for i in unslashed_and_active_indices
        if (i in store.latest_messages
            and i not in store.equivocating_indices
            and get_ancestor(store, store.latest_messages[i].root, store.blocks[root].slot) == root)
    ))
    if store.proposer_boost_root == Root():
        # Return only attestation score if ``proposer_boost_root`` is not set
        return attestation_score
```

All validators eligible to vote

Votes are weighted by EB

Filter on validators whose latest vote was for a block whose ancestor is `root`

LMD GHOST fork choice



Ignoring undesirable blocks

- Was it correctly signed by the expected proposer?
  - Actually checked upstream of fork choice (gossip layer?)
- Are all its ancestor blocks valid?
- Is the block's post-state hash correct?
  - That is, does the block respect the STF (State Transition Function)?

New in Deneb:

- Is its data available?
  - Have we seen/can we get its attached blobs?

Potential future:

- Is the block non-censoring?
- More sophisticated data availability (e.g. PeerDAS - see Fulu)

Ignoring undesirable blocks

```python
def on_block(store: Store, signed_block: SignedBeaconBlock) -> None:
    """
    Run ``on_block`` upon receiving a new block.
    """
    block = signed_block.message
    # Parent block must be known
    assert block.parent_root in store.block_states
    # Blocks cannot be in the future. If they are, their consideration must be delayed until they are in the past.
    assert get_current_slot(store) >= block.slot

    # Check that block is later than the finalized epoch slot (optimization to reduce calls to get_ancestor)
    finalized_slot = compute_start_slot_at_epoch(store.finalized_checkpoint.epoch)
    assert block.slot > finalized_slot
    # Check block is a descendant of the finalized block at the checkpoint finalized slot
    finalized_checkpoint_block = get_checkpoint_block(
        store,
        block.parent_root,
        store.finalized_checkpoint.epoch,
    )
    assert store.finalized_checkpoint.root == finalized_checkpoint_block

    # [New in Deneb:EIP4844]
    # Check if blob data is available
    # If not, this block MAY be queued and subsequently considered when blob data becomes available
    # *Note*: Extraneous or invalid Blobs (in addition to the expected/referenced valid blobs)
    # received on the p2p network MUST NOT invalidate a block that is otherwise valid and available
    assert is_data_available(hash_tree_root(block), block.body.blob_kzg_commitments)
```

Are all its ancestors viable?

Is not from the future

Is not too old

Data is available

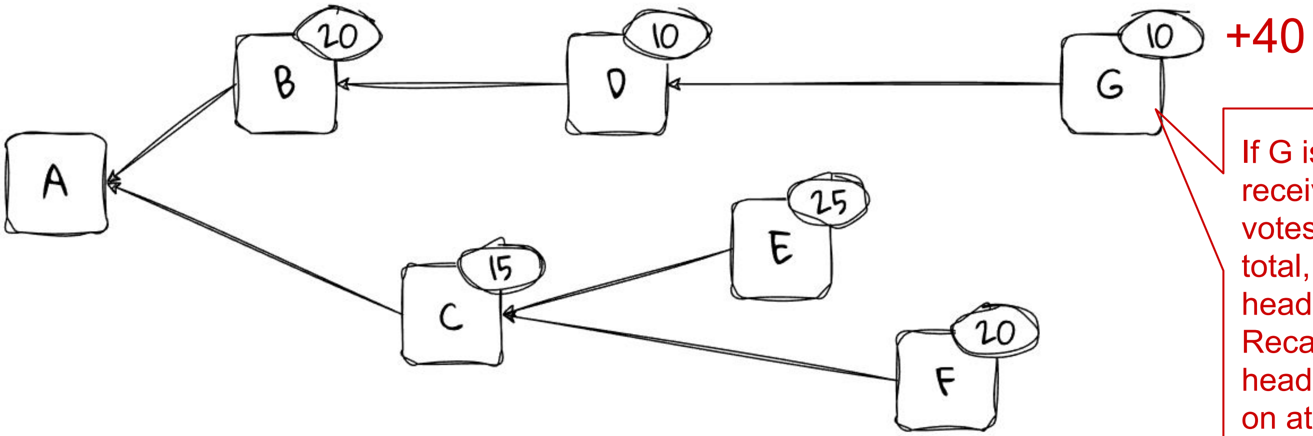
Deneb
[fork-choice.md]()

Proposer Boost

- In 2020 a “balancing attack” was identified
  - A (very) small number of validators could fork the chain into two equally weighted branches and maintain this indefinitely.
- The defence against this is “Proposer Boost”
  - Blocks received in a timely way (within 4 seconds of slot start) are assigned a huge extra weight in the `get_weight()` calculation.
    - The extra weight almost certainly makes the timely block the head of the chain
  - This makes honest proposers almost sure to build on a timely block that they received
  - “A block that is timely should not expect to be reorged”

Book

Proposer Boost and fork choice



If G is timely it receives extra votes of 40% of total, making it the head block. Recall that E is head based only on attestation weight

Why 40?

Proposer Boost in code

get_weight()

Dummy value

No timely block was received

```
    if store.proposer_boost_root == Root():
        # Return only attestation score if ``proposer_boost_root`` is not set
        return attestation_score

    # Calculate proposer score if ``proposer_boost_root`` is set
    proposer_score = Gwei(0)
    # Boost is applied if ``root`` is an ancestor of ``proposer_boost_root``
    if get_ancestor(store, store.proposer_boost_root, store.blocks[root].slot) == root:
        proposer_score = get_proposer_score(store)
    return attestation_score + proposer_score
```

```
def get_proposer_score(store: Store) -> Gwei:
    justified_checkpoint_state = store.checkpoint_states[store.justified_checkpoint]
    committee_weight = get_total_active_balance(justified_checkpoint_state) // SLOTS_PER_EPOCH
    return (committee_weight * PROPOSER_SCORE_BOOST) // 100
```

40

Phase 0

fork-choice.md

Add 40% of the total stake attesting in this slot to the weight of the branch with the timely block.

Proposer Boost in code

A timely block is received within 4s of the slot start

on_block()

```
# Add block timeliness to the store
time_into_slot = (store.time - store.genesis_time) % SECONDS_PER_SLOT
is_before_attesting_interval = time_into_slot < SECONDS_PER_SLOT // INTERVALS_PER_SLOT
is_timely = get_current_slot(store) == block.slot and is_before_attesting_interval
store.block_timeliness[hash_tree_root(block)] = is_timely

# Add proposer score boost if the block is timely and not conflicting with an existing block
is_first_block = store.proposer_boost_root == Root()
if is_timely and is_first_block:
    store.proposer_boost_root = hash_tree_root(block)
```

Record the timely block for use in get_weight()

Phase 0

fork-choice.md

on_tick_per_slot()

```
# If this is a new slot, reset store.proposer_boost_root
if current_slot > previous_slot:
    store.proposer_boost_root = Root()
```

Dummy value

Forking out late blocks with Proposer Boost

- Proposer Boost solves the balancing attack, but also gives us the opportunity to strongly discourage late block publication.
  - Honest proposers should publish a block by 4s to allow time for attestations to be made and to be collected by the network.
  - However, the spec allows a block received as late as 12s to be built upon by the next proposer.
  - Block builders (MEV searchers) took advantage of this to publish later and later with low risk of being orphaned.
  - This destabilises the network as such late blocks gather few attestations, and it penalises timely attesters (who attest to an empty slot)
- Proposer Boost allows an honest proposer to build on the *parent* of a late block, its block becoming the head (since it is timely)
  - Thus the late block is orphaned / re-orged out of the chain
  - See `get_proposer_head()`

Slashing in LMD GHOST

Slashing solves the “nothing at stake” problem with proof of stake.

Equivocation is costless:

- It’s costless for proposers to publish multiple blocks (unlike in PoW)
  - E.g. why not build on every branch?
- It’s costless for validators to vote for everything they see
  - A rich source of fork choice attacks.

Solution:

- Slash equivocating proposers and equivocating attesters

January 2014: [Slasher: A Punitive Proof-of-Stake Algorithm]

Yay! We have ourselves a consensus mechanism 🥳

- Timescale:
  - Slot-based: 12s
- Goal:
  - Used by the block proposer to decide on which branch to build its block
  - Used by attesters to choose which branch to attest to => convergence
- Heuristic:
  - Which branch is least likely to be orphaned in future?
- Based on:
  - “Weighing” the votes received in attestations
  - Note that a maximum of only 3.125% of the votes are “fresh” (same-slot)
- Properties
  - Liveness: it will always output a viable head block on which to build (proof under synchrony)
  - Safety: no useful guarantees (however, see the confirmation rule work.)

But we can do better 🤔

Under proof of stake:

1) We have a well-defined set of participants
   - as in classical consensus, unlike PoW
2) We can apply economic penalties for bad behaviour
   - slashing

How about we use these to add some stronger safety properties to our chain?

Casper FFG

Naming

- Casper
  - Kind of influenced by Vlad Zamfir's Casper protocol that uses GHOST
    - see Casper the Friendly Ghost…
  - But Casper FFG has very little similarity to Zamfir's protocols and doesn't use GHOST 🤷‍♂️
- FFG
  - The "Friendly Finality Gadget"
    - A "gadget" in Vitalik-speak is a self-contained enhancement to another process
    - So, Casper FFG is a gadget for adding finality to an existing blockchain consensus mechanism
    - Was originally planned to add finality to PoW as part of the PoS transition (EIP-1011)

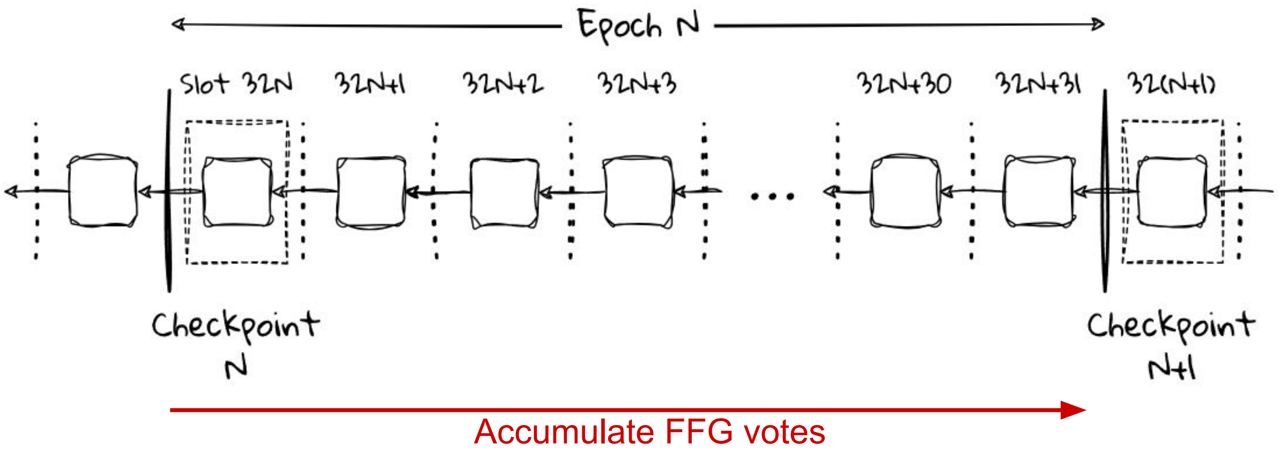
Original Casper FFG Paper

Casper FFG Overview

- Timescale:
  - Epoch-based: 32 slots / 6.4 minutes / 384 s
- Goal:
  - Confer “finality” on the chain: a checkpoint that will never be reverted (except at enormous cost)
- Heuristic:
  - Two-phase commit based on agreement among validators having at least ⅔ of the stake
- Based on:
  - “Weighing” the source and target votes received in attestations contained in blocks
- Properties
  - Plausible Liveness: cannot get into a stuck state that is unable to finalise anything
  - Accountable Safety: finalising conflicting checkpoints comes at enormous cost
    - aka Economic Finality

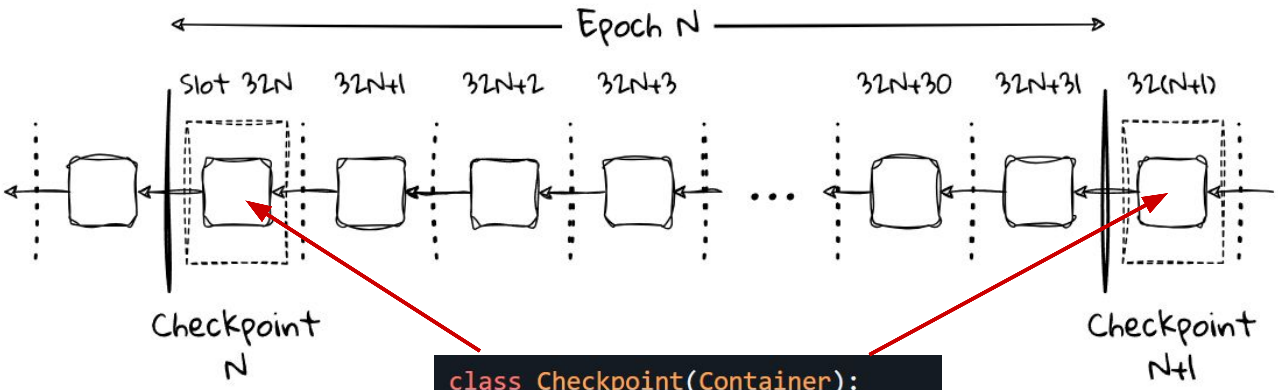
Checkpoints

- Casper FFG relies on seeing votes from the whole validator set.
- This is too expensive to do every slot, so we accumulate votes across an epoch.
- 1/32 of the validator set votes at each slot.
- Accounting is done at each epoch end transition.



Checkpoints

- A checkpoint is the block at the first slot of the Epoch
  - Or the previous block if that slot is empty

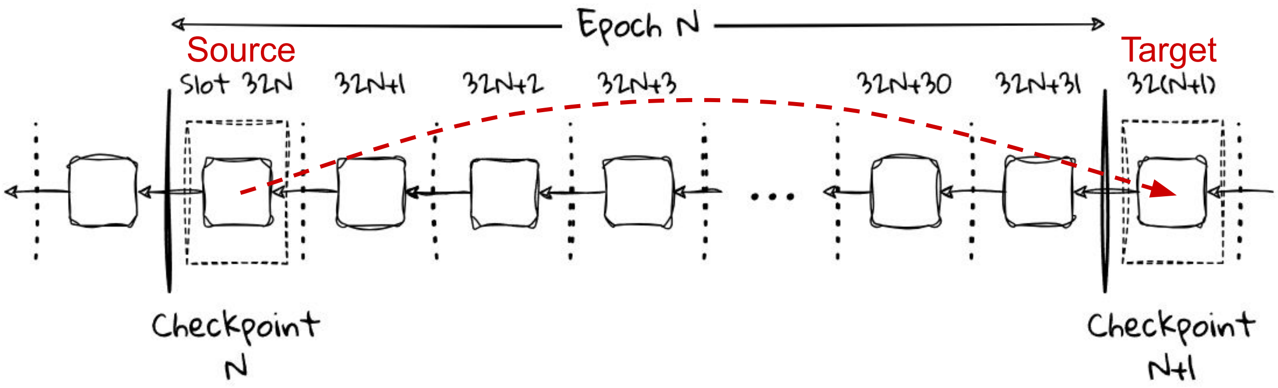



```python
class Checkpoint(Container):
    epoch: Epoch
    root: Root
```

Checkpoints

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint
```

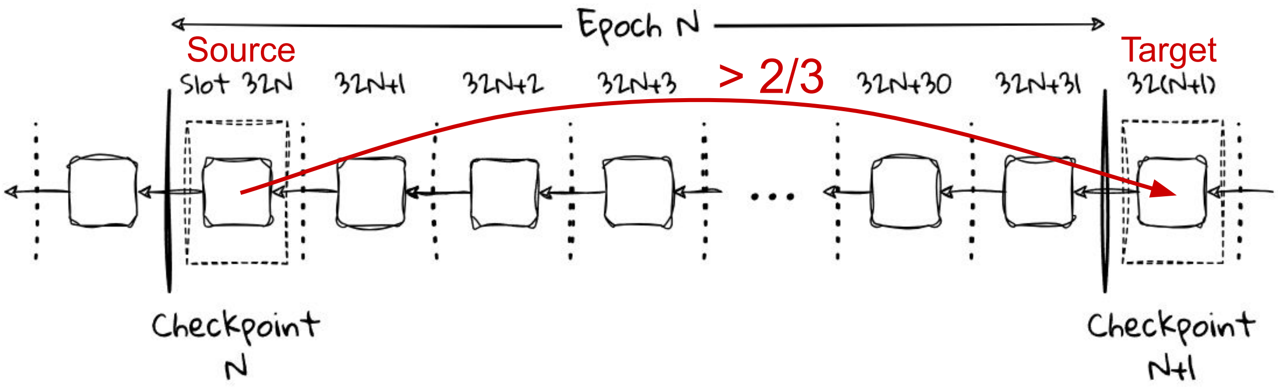
- Validators vote for a source and a target checkpoint
  - Source: the highest justified checkpoint I know of.
  - Target: the checkpoint I see in the current epoch.



Supermajority links

- A supermajority link occurs when >⅔ of validators (by stake) have made the same source → target vote

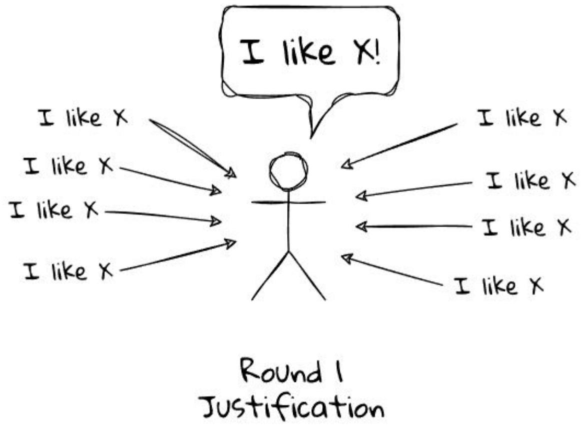
```python
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint
```



Two-phase commit: justification

- My `target` vote says “My preferred branch has current checkpoint X”.
- If I hear that ⅔ of validators agree with me, I locally mark X *justified*.
    - I’ve seen a supermajority link with X as target

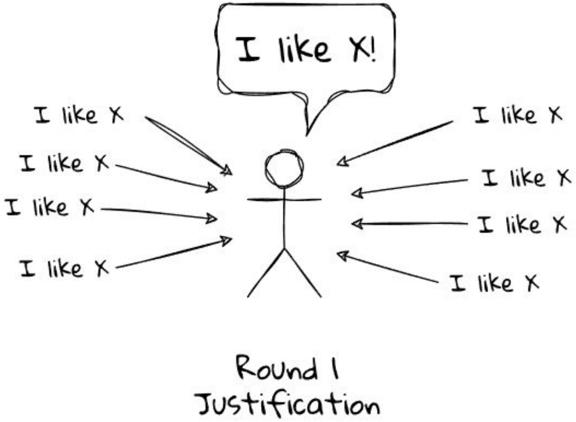
Target votes



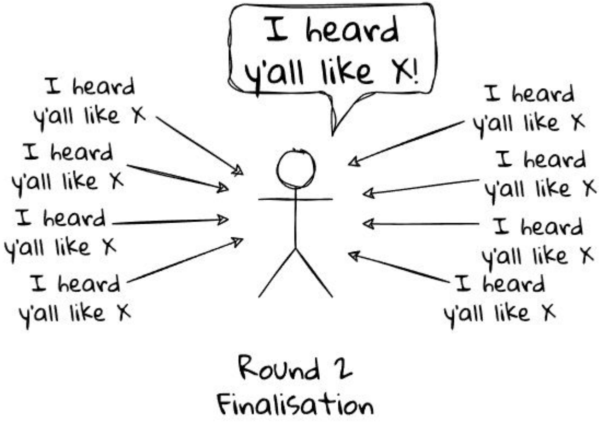
Two-phase commit: finalisation

- My `source` vote says “I heard that >⅔ of you like X so I justified it”
- If I hear that ⅔ of validators also said that, I locally mark X as *finalised*.
  - I’ve seen a supermajority link with X as source
  - We know that ⅔ of validators have justified X and will therefore never revert it

Target votes



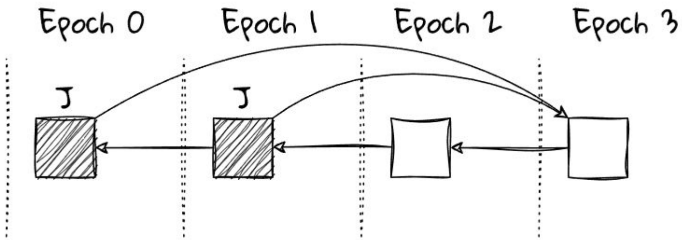
Source votes



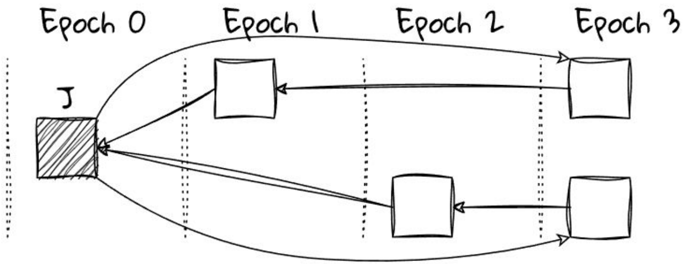
Casper Commandment: no double vote

A validator must not publish distinct votes $s_1 \rightarrow t_1$ and $s_2 \rightarrow t_2$ such that $h(t_1) = h(t_2)$

That is, a validator must make at most one vote for any target epoch.



No!

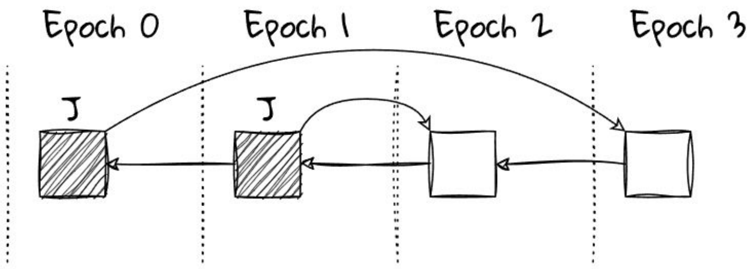


No!

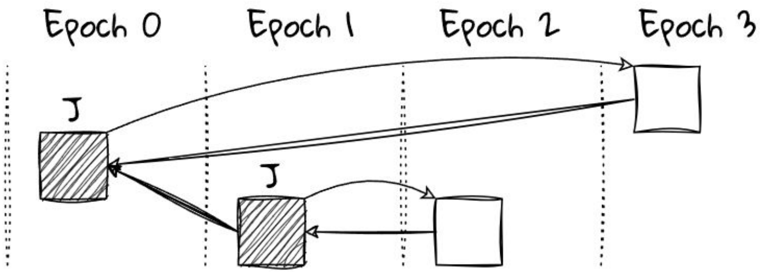
Casper Commandment: no surround vote

A validator must not publish distinct votes $s_1 \to t_1$ and $s_2 \to t_2$ such that $h(s_1) < h(s_2) < h(t_1) < h(t_2)$

That is, a validator must not make a vote such that its link either surrounds, or is surrounded by, a previous link it voted for.



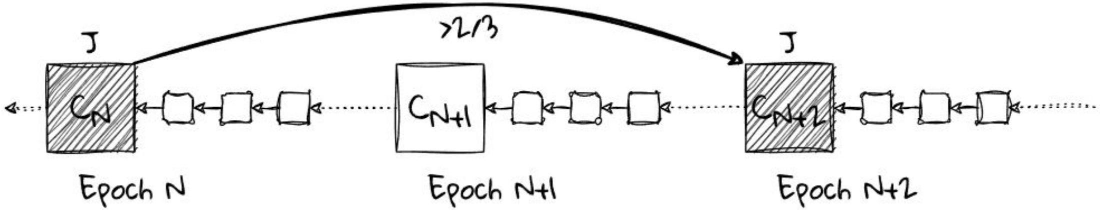
No!



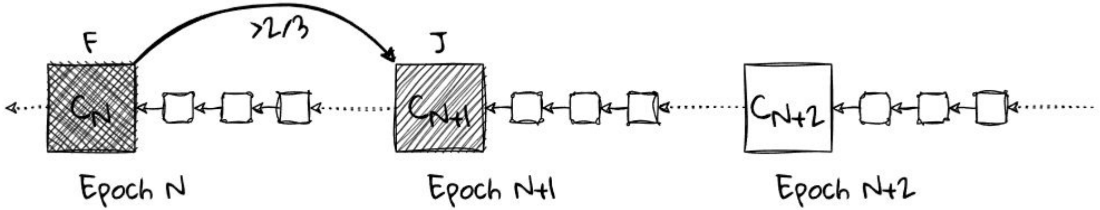
No!
(See Holesky)

Accountable Safety and Plausible Liveness

- The proofs of Accountable Safety and Plausible Liveness rely on these two commandments.
- And one further rule that we only finalise a checkpoint that is the direct child of a justified checkpoint



Justification (local)



Finalisation (global)

Except, see k-finality

Slashing in Casper FFG

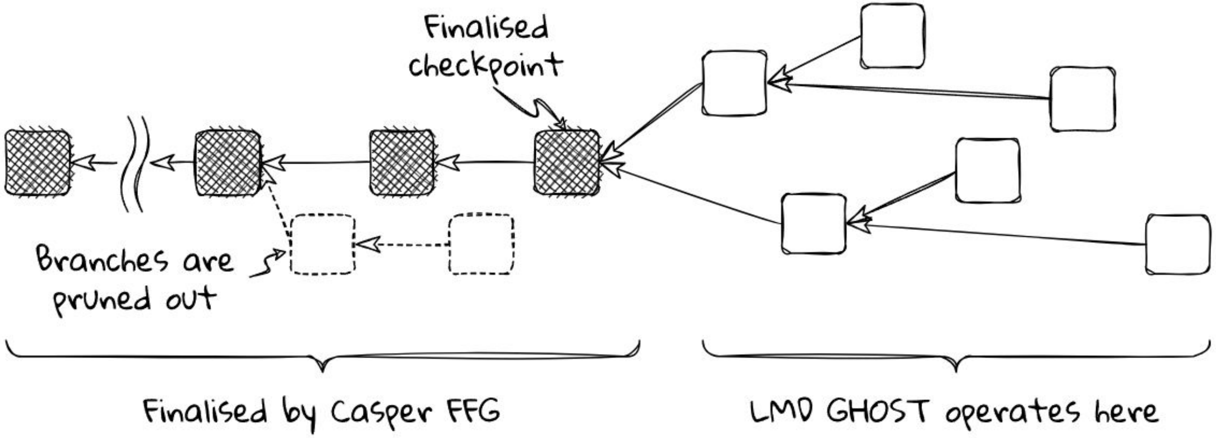
Delivers “economic finality”: we slash validators that break a commandment

- If <⅓ of validators is adversarial, safety is guaranteed (as with PBFT)
- If >⅓ of validators acts so as to finalise conflicting checkpoints:
    - At least ⅓ of validators must have broken a commandment.
    - We can detect this and prove it onchain!
    - The aggregate cost to the attacker would be at least ⅓ of the total staked Ether
    - Thus, we have “economic finality” - a quantifiable cost to messing with finality, even if the attacker has more than ⅓ of the validators

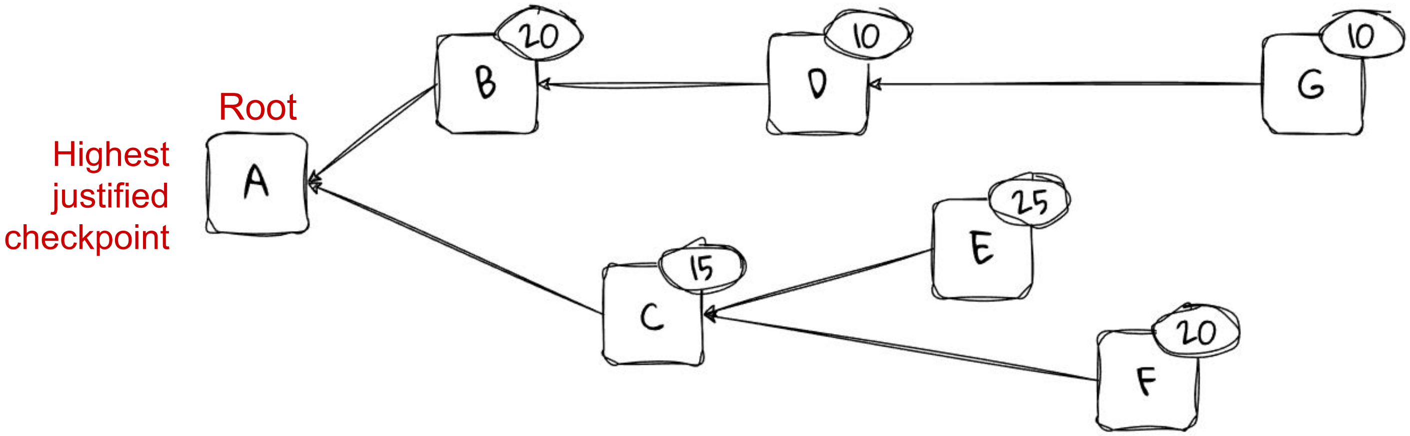
This is a significant feature: classical consensus makes no guarantees if >⅓ of replicas are adversarial / faulty / Byzantine. BFT algorithms will also halt if >⅓ of replicas are offline, whereas Casper FFG continues trying to justify checkpoints.

GASPER

Applying the FFG to LMD GHOST



Modified LMD GHOST fork choice



Apply Casper FFG's fork choice: "follow the chain containing the justified checkpoint of the greatest height".

get_head()

```python
def get_head(store: Store) -> Root:
    # Get filtered block tree that only includes viable branches
    blocks = get_filtered_block_tree(store)
    # Execute the LMD-GHOST fork choice
    head = store.justified_checkpoint.root
    while True:
        children = [
            root for root in blocks.keys()
            if blocks[root].parent_root == head
        ]
        if len(children) == 0:
            return head
        # Sort by latest attesting balance with ties broken lexicographically
        # Ties broken by favoring block with lexicographically higher root
        head = max(children, key=lambda root: (get_weight(store, root), root))
```

Apply Casper FFG fork choice

Follow the branch with the maximum weight

Weigh the branch rooted at block "`root`"

Phase 0
fork-choice.md

The world were that the end of the story…

Gasper in reality

Interaction bugs. The "interface" between Casper FFG finalization and LMD GHOST fork choice is a source of significant complexity, leading to a number of attacks that have required fairly complicated patches to fix, with more weaknesses being regularly discovered.

Vitalik

How LMD GHOST and Casper FFG are actually bolted together

Issue one: block tree filtering

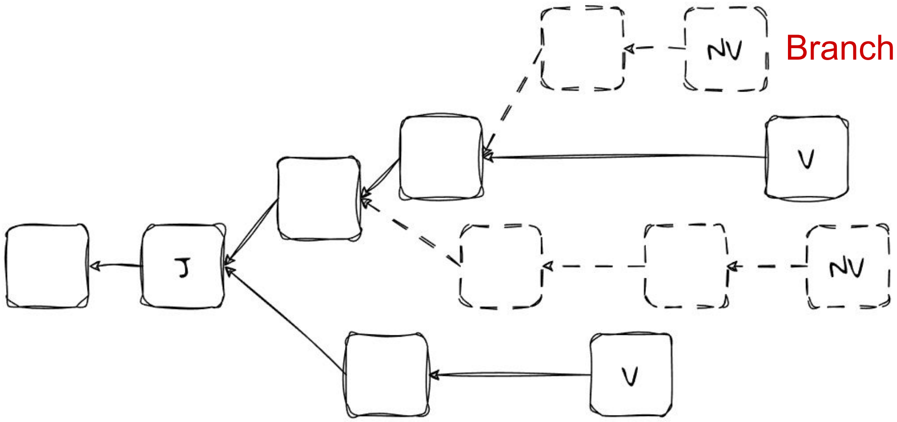
```python
def get_head(store: Store) -> Root:
    # Get filtered block tree that only includes viable branches
    blocks = get_filtered_block_tree(store)
    # Execute the LMD-GHOST fork choice
    head = store.justified_checkpoint.root
    while True:
        children = [
            root for root in blocks.keys()
            if blocks[root].parent_root == head
        ]
        if len(children) == 0:
            return head
        # Sort by latest attesting balance with ties broken lexicographically
        # Ties broken by favoring block with lexicographically higher root
        head = max(children, key=lambda root: (get_weight(store, root), root))
```

WTF is this???

Phase 0
fork-choice.md

Issue one: block tree filtering

- `filter_block_tree()` removes from fork choice consideration any branches with “non-viable heads”.



Branch is removed from consideration

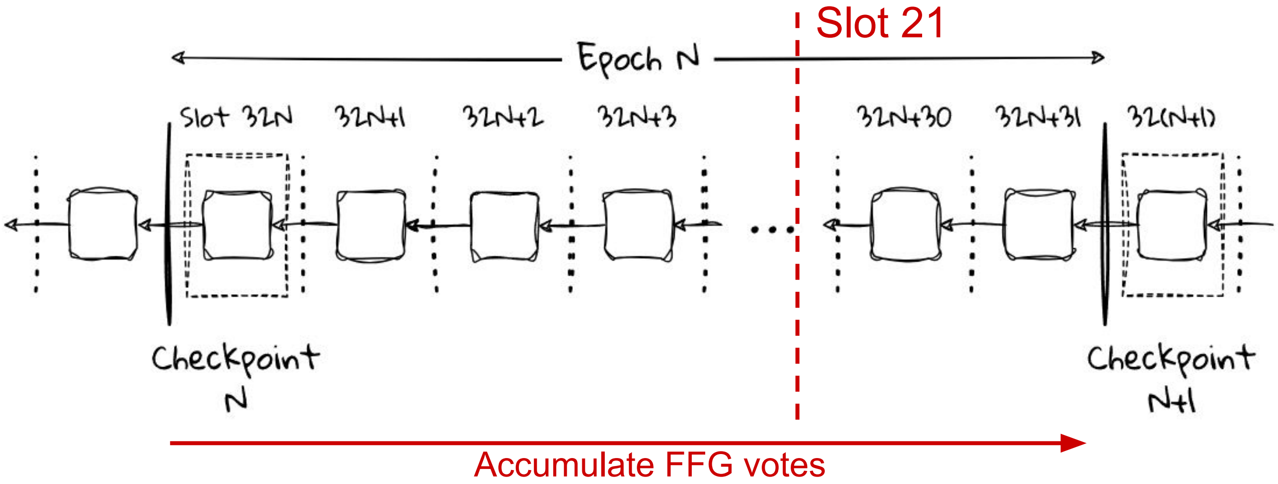
Branch is removed from consideration

Issue one: block tree filtering

- Non-viable means that the block's associated state does not agree with my Store about the current justified and finalised checkpoints.
- Resolves a potential finalisation deadlock issue.
    - Plausible Liveness failure.
- Originates from a conflict between LMD GHOST's fork choice and Casper FFG's fork choice:
    - Casper's rule is "start from the highest justified checkpoint"
    - How to handle this in LMD GHOST when different blocks have different views of the highest justified checkpoint?

Issue two: unrealised justification and finalisation

- Justification and finalisation accounting is done at the end of each epoch.
- However, by slot 21 (⅔ of the way through the epoch) I might have seen enough votes to form a supermajority link: *unrealised* justification



Issue two: unrealised justification and finalisation

- Unrealised justification reorg attack
    - allows the proposer of the first block of an epoch to fork out up to nine blocks from the end of the previous epoch.
- Justification withholding attack
    - adversary can reorg arbitrary numbers of blocks at the start of an epoch.
- Main issue:
    - Justification is updated in the beacon state only at the end of an epoch
    - Filter block tree was too aggressive because it did not account for “unrealised justification”
        - Blocks that would match my checkpoints if they had gone through epoch processing
        - Marked as non-viable, while actually being viable, and being removed from fork choice
            - Leads to re-orgs.

Unrealised Justification

The world when we have single slot finality