Manual Proof of Deadlock Freedom for the Ethereum Fork Choice specification

Roberto Saltini Chenyi Zhang
Dependable Distributed Systems Team, ConsenSys R&D
{name}.{surname}@consensys.net

Spec version

https://github.com/ethereum/consensus-specs/pull/3290

Assumptions

1. Hash functions are ideal (i.e. no collision is possible)
2. Honest validators behave as follows when they receive a message `m` (i.e block, attestation or attester_slashing).

    1. Add message `m` to the set `M` (which is a set of messages initialized to empty)
    2. `while` (is there a fork choice handler function that can be called on a message in set `M` that does not raise any assertion)
       {
       2.1. `let (m, h)` be a message `m` in `M` and a fork choice handler function, respectively, such that calling `h` on `m` does not raise any assertion.
       2.2. `h(store, m)`
       2.3. remove `m` from `M`
       }

3. Honest nodes do not discard any attestation that they receive regardless of how old it is. In other words, the `validate_target_epoch_against_current_time(store, attestation)` function in the current specs should be regarded as a no-op
4. Honest nodes send `AttesterSlashing` messages for any couple of slashing attestations that they detect.
5. Honest stake is > ⅔ the total stake
6. No bound on the amount of attestations that can be included in a block

Property to Prove

A distributed system running under the assumptions listed above can never reach a state where it is impossible to finalize a new epoch.

Definitions and Notation

1. For any map `m`, let `m.Keys` be the set including all the non-empty keys of map `m`
2. For any map `m` such that `m.Keys ⊆ store.blocks.Keys`, let `fork_choice_map_filter(store, m)` be the subset map of map `m` including only and all the keys `k` such that `store.blocks[k].slot >= store.finalized_checkpoint.slot`.
3. We say that two honest nodes are *fork-choice equivalent* if the following fields of their respective stores have the same value:
    a. `justified_checkpoint`
    b. `fork_choice_map_filter(store, blocks)`
    c. `fork_choice_map_filter(store, block_states)`
    d. `fork_choice_map_filter(store, unrealized_justifications)`
    e. `finalized_checkpoint`
    f. `checkpoint_states[store.justified_checkpoint]`
    g. `equivocating_indices`
    h. `proposer_boost_root`
    i. `time`
    j. `genesis_time`
    k. `latest_messages` for any index not in `equivocating indices`
4. Let `post_state(B)` be the state obtained by executing the beacon chain `state_transition` function on all the blocks of the chain headed by `B` starting from the genesis block.
5. Let `M(state: BeaconState, other_parameters)` be a method that modifies the `BeaconState` state.
   We use the writing `M(state, other_parameters).` to refer to the modified state.
   For example, `process_slots(state, current_slot).current_justified_checkpoint` corresponds to the value of the field `current_justified_checkpoint` after the state has been modified according to the execution of `process_slots`.
6. Let `first_slot(t) == true`, where `t` is any time, iff the slot associated with time `t` is the first slot of an epoch.
7. If t is a time, let `epoch(t)` be the epoch associated with time `t`.
8. If t is a block, let epoch(B) == compute_epoch_at_slot(B.slot)
9. `store(v)` corresponds to the store of the honest validator `v`
10. `[N1, N2]` corresponds to the set `{x \in Z | N1 <= x <= N2}`
11. `[N1, N2[` corresponds to the set `{x \in Z | N1 <= x < N2}`

Proof

Outline

The proof follows the following outline

- Prove that after any period of asynchrony, there exists a schedule involving only honest validators that lead to all honest validators being fork-choice equivalent
- Prove that there exists a schedule starting from a point where all honest validators are fork-choice equivalent such that honest validators keep continuously extending and attesting the same chain.
- Leverage on the poofs above to prove deadlock freedom.

Detailed Proof

Note: Given that GDoc does not allows numbering Lemmas automatically (at least, as far as I know it), Lemmas have been numbered in increments of 10 to allow preserving the logical progression of the proof argument according to Lemmas' ascending numbering in the case that new Lemmas need to be added or existing Lemmas need to be reordered.

Lemma 30
The blockchain is <⅓-slashable.

Proof.
Direct consequence of Assumption 5 and the property that honest validators never slash themselves.

Q.E.D.

Lemma 35.
For any honest validator `v`, `store(v).justified_checkpoint.epoch` is monotonically increasing with time, i.e., let `store(v).justified_checkpoint(t)` be the value of `store(v).justified_checkpoint` at time `t`, then for any two times `t < t'`, we have that `store(v).justified_checkpoint(t).epoch <= store(v).justified_checkpoin(t').epoch`.

Proof.
Any location in the code where `store(v).justified_checkpoint.epoch` is set to a value `x`, it is before checked that `x > store(v).justified_checkpoint.epoch`.

Q.E.D.

Lemma 40.
Let `M` be any set of messages, and `v` and `v'` be any two honest validators.

If by the time `v` and `v’` execute `on_tick_per_slot(, )` in the first slot of the next epoch, i) they have both received all and only the messages in the set `M` and ii) their clocks are perfectly synced, then regardless of the order (time) when they received any of the messages in `M`, after executing `on_tick_per_slot(, )` `v` and `v’` will be fork-choice equivalent with the only exception of the `latest_messages` fields.

Proof.
In order to prove the Lemma, we need to establish that some other Store fields, aside from those directly involved in the definition of fork-choice equivalence, have the same value for `v` and `v’` after they execute `on_tick_per_slot(, )` in the first slot of the next epoch.

Let us now look at each field of the Store data structure involved in proving the Lemma. (We label those directly involved in the definition of fork-choice equivalence with the notation [FCE])

1. [FCE] `time`, `genesis_time`
   Obviously the same.
2. `unrealized_justified_checkpoint`
   Every time that `on_block(, B)` is executed, it executes `compute_pulled_up_tip(store, block_root)` which sets `store.unrealized_justified_checkpoint` to the highest known unrealized justified checkpoint so far.
   By Assumption 2, both `v` and `v’` will set their respective `store.unrealized_justified_checkpoint` to the highest unrealized justified checkpoint according to the messages in the set `M`.
   Finally, as a consequence of Lemma 30, the highest unrealized justified checkpoint according to the messages is unique.
3. `unrealized_finalized_checkpoint`
   Every time that `on_block(, B)` is executed, it executes `compute_pulled_up_tip(store, block_root)` which sets `store.unrealized_finalized_checkpoint` to the highest known unrealized finalized checkpoint so far.
   By Assumption 2, both `v` and `v’` will set their respective `store.unrealized_finalized_checkpoint` to the highest unrealized finalized checkpoint according to the messages in the set `M`.
   Finally, as a consequence of Lemma 30, the highest unrealized finalized checkpoint according to the messages is unique.
4. [FCE] `finalized_checkpoint`
   Every time that `on_block(, B)` is executed, it sets `store.finalized_checkpoint` to the highest known realized finalized checkpoint so far.
   When `on_tick_per_slot(, )` is executed in the first slot of the next epoch, `update_checkpoints(store, state.current_justified_checkpoint, state.finalized_checkpoint)` is executed which sets

`store.finalized_checkpoint` to
`store.unrealized_finalized_checkpoint` if
`store.unrealized_finalized_checkpoint.epoch >`
`store.finalized_checkpoint`.
Given that on the first slot of the next epoch
`store.unrealized_finalized_checkpoint` corresponds to the highest realized finalized checkpoint, after `on_tick_per_slot(, )` is executed,
`store.finalized_checkpoint` corresponds to the highest known realized finalized checkpoint.
By Assumption 2 and point 3 above, both `v` and `v'` will set their respecti*ve*
`store.finalized_checkpoint` to the highest realized finalized checkpoint according to the messages in the set `M`.
Finally, as a consequence of Lemma 30, the highest finalized checkpoint according to the messages is unique.

5. [FCE] `justified_checkpoint`
Every time that `on_block(, B)` is executed, it sets
`store.justified_checkpoint` to the highest known realized justified checkpoint so far.
When `on_tick_per_slot(, )` is executed in the first slot of the next epoch,
`update_checkpoints(store, state.current_justified_checkpoint,`
`state.finalized_checkpoint)` is executed which sets
`store.justified_checkpoint` to
`store.unrealized_justified_checkpoint` if
`store.unrealized_justified_checkpoint.epoch >`
`store.justified_checkpoint`.
Given that on the first slot of the next epoch
`store.unrealized_justified_checkpoint` corresponds to the highest realized justified checkpoint, after `on_tick_per_slot(, )` is executed,
`store.justified_checkpoint` corresponds to the highest known realized justified checkpoint.
By Assumption 2 and point 2 above, both `v` and `v'` will set their respective
`store.justified_checkpoint` to the highest realized justified checkpoint according to the messages in the set `M`.
Finally, as a consequence of Lemma 30, the highest justified checkpoint according to the messages is unique.
6. [FCE] `fork_choice_map_filter(store, blocks)`
Due to the points 1 and 4 above, and Assumption 2, any block `B` such that B.slot >= finalized_checkpoint.slot that is added to the store of `v` is also added to the store of `v'` as well, and vice versa. Additionally, due to Assumption 1, each key of the map `blocks` of the store of either `v` or `v'`, once set to a block `B`, is never updated to a block different from `B`. This concludes the proof for this field.

7. [FCE] `fork_choice_map_filter(store, block_states)`
   Every time that a block `B` is added to the store, its post-state, resulting from executing the function `state_transition(store.block_states[B.parent_root], B, true)`, is added to the `block_states` field. Point 6 above, the fact that `state_transition` is a deterministic function and assumption 1 then conclude the proof for this field.
8. [FCE] `fork_choice_map_filter(store, unrealized_justifications)`
   Every time that `on_block(, B)` is executed, it executes `compute_pulled_up_tip(store, block_root)` which sets `store.unrealized_justifications[B]` to the highest justified checkpoint according to all the attestations included in the chain headed by `B`.
   Assumption 1 and 2, point 7 above and the fact that the value of `store.unrealized_justifications[B]` is uniquely determined by `B`, after executing `on_tick_per_slot(, )`, conclude the proof for this field.
9. [FCE] `proposer_boost_root`.
   Set to `Root()` for both `v` and `v'` in on_tick_per_slot
10. [FCE] `checkpoint_states[store.justified_checkpoint]`
    Due to the definition of `state_transition`, both validators have received at least a block, say `B`, with an attestation with target `store.justified_checkpoint`.
    Therefore, both validators have executed `store_target_checkpoint_state(_, store.justified_checkpoint)`.
    Given that
    a. `store.block_states` is the same for `v` and `v'`
    b. any other function executed in `store_target_checkpoint_state` is a deterministic function of `store.block_states` and `store.justified_checkpoint`

    `checkpoint_states[store.justified_checkpoint]` is the same for `v` and `v'`
11. [FC] `equivocating_indices`
    `v` and `v'` receive the same set of `AttesterSlashing` messages.
    Hence, `v` and `v'` will detect the same set of `equivocating_indices`

Q.E.D.

Lemma 50.
Let `M` be any set of messages, `v` and `v'` be any two honest validators, and `e` be the current epoch.
If i) by the time `v` and `v'` execute `on_tick_per_slot(, )` in the first slot of the epoch `e+1`, i.a) they have both received all and only the messages in the set `M` and i.b) their clocks are perfectly synced, and ii) during epoch `e+1` ii.a) message latency is 0, ii.b) their clocks keep

being perfectly synced and ii.c) dishonest validators do not send any messages, then after executing `on_tick_per_slot(, )` in the first slot of the epoch `e+2`, `v` and `v′` will be fork-choice equivalent.

Proof.
From Lemma 40 it follows that by the time `v` and `v′` execute `on_tick_per_slot(, )` in the first slot of epoch `e+1`, they are fork-choice equivalent with the only exclusion of `latest_messages`. Following the reasoning outlined in Lemma 40, one can show that, if in epoch `e+1` i) message latency is 0, ii) the clocks of `v` and `v′` are perfectly synced and iii) dishonest nodes do not send any message, then after `v` and `v′` execute `on_tick_per_slot(, )` in the first slot of epoch e+2, they will be fork-choice equivalent with exclusion of `latest_messages`.

Let us now prove that after `v` and `v′` execute `on_tick_per_slot(, )` in the first slot of epoch `e+2`, they will also have the same value of the field `store.latest_messages` for any index not in `equivocating_indices`.

Due to Assumption 4, by the end of epoch `e+1`, all equivocating (slashable) attestations included in the set of messages `M` will be detected, the corresponding `AttesterSlashing` messages sent and received by any honest node. Also, given that only honest nodes are active during epoch `e+1`, no further slashable attestations will be sent.

Hence, by the end of epoch `e+1`, if a dishonest validator `b` has sent more than one attestation for the same target epoch, a corresponding `AttesterSlashing` message will be received by any honest node and as a consequence of this, any honest node will add the index associated with `b` to the set of equivocating indices.
This proves that for any given epoch `ea` and any validator `x` whose index is not in equivocating indices, at most one attestation has been sent by validator `x` with target `ea`.

Due to Assumption 3, `validate_on_attestation` does not depend on `store.time`.
Due to Assumption 2, any attestation `A` that has been received by `v` (i.e. `on_attestation(_, A, _)` was executed without raising an assertion) is also received by `v′` by the time it executes `on_tick_per_slot(, )` in the first slot of the epoch `e+2`, and vice versa.
By following a reasoning similar to the one outlined in Lemma 40 for `checkpoint_state`, one can conclude that `v` and `v′` compute the same value for `target_state` inside `on_attestation` when they receive attestation `A`.
Given i) that we only considering the indices of `latest_messages` that are not in `equivocating_indices`, ii) that `latest_messages` only keeps the message with highest target epoch and iii) that, as proved above, for a target epoch there exits only one attestation sent by a validator whose index is not in `equivocating indices`, we can conclude that after `v` and `v′` executes `on_tick_per_slot(, )` in the first slot of epoch `e+2`, they will also have the same value of the field `store.latest_messages` for any index not in

`equivocating_indices`.

Q.E.D.

Lemma 60.
If two honest nodes are fork-choice equivalent, their respective executions of `get_head` return the same value.

Proof.
Obvious from the fact that `get_head` only depends on the fields included in the definition of fork-choice equivalence.

Q.E.D.

Lemma 74.
For any honest validator `v` and block `B` in `store(v).unrealized_justifications`, `store(v).unrealized_justifications[B].epoch <= store(v).unrealized_justified_checkpoint.epoch`.

Proof.
The only place where `store(v).unrealized_justifications` is modified is in the execution of `compute_pulled_up_tip`.
Moreover, the only element of the map `store(v).unrealized_justifications` that is modified is the one with key `block_root`, where `block_root` is an input parameter to `compute_pulled_up_tip`.
Observe that `compute_pulled_up_tip` always executes `update_unrealized_checkpoints(store(v), state.current_justified_checkpoint, state.finalized_checkpoint)` with `state.current_justified_checkpoint == store(v).unrealized_justifications[block_root]`.
As a consequence of this, `store(v).unrealized_justified_checkpoint.epoch` is set to `max(store(v).unrealized_justified_checkpoint.epoch, store(v).unrealized_justifications[block_root].epoch)`.
This concludes the proof.

Q.E.D.

Lemma 75.
For any honest validator `v`, there exists a block `B` such that `store(v).unrealized_justified_checkpoint == store(v).unrealized_justifications[B]`.

Proof.

The only place where `store(v).unrealized_justified_checkpoiont` is modified is in the execution of `update_unrealized_checkpoints(store(v), unrealized_justified_checkpoint, unrealized_finalized_checkpoint)` where, if set, it is set to `unrealized_justified_checkpoint`.
The only place where `update_unrealized_checkpoints` is called is in `compute_pulled_up_tip` where the parameter `unrealized_justified_checkpoint` corresponds to `store(v).unrealized_justifications[block_root]` where `block_root` is a parameter of `compute_pulled_up_tip`.

This and the fact that `store(v).unrealized_justifications` is only modified in the execution of `compute_pulled_up_tip` conclude the proof.

Q.E.D.

Lemma 76.
For any validator `v`, let `store(v)(B)` be the value of `store(v)` after `v` receives block `B`, i.e. after it executes `on_block(store(v), B)` without raising exceptions.
If `get_voting_source(store(v)(B), B).epoch > store(v).justified_checkpoint.epoch`, then `get_voting_source(store(v)(B), B).epoch == store(v)(B).justified_checkpoint.epoch`.

Proof.
Let us consider two cases.

1. `epoch(B) < current_epoch`
   In this case, `get_voting_source(store(v)(B), B) == store(v)(B).unrealized_justifications[B]`.
   Therefore, `store(v)(B).unrealized_justifications[B].epoch > store(v).justified_checkpoint.epoch`.
   Observe that `compute_pulled_up_tip` (which is executed by `on_block`) executes `update_checkpoints(store, state.current_justified_checkpoint, state.finalized_checkpoint)` with `state.current_justified_checkpoint == store(v)(B).unrealized_justifications[B]` which sets `store(v)(B).justified_checkpoint.epoch` to `max(store(v).justified_checkpoint.epoch, store(v)(B).unrealized_justifications[B].epoch)`.
   This concludes the proof of the Lemma for this case.
2. `epoch(B) == current_epoch`
   In this case, `get_voting_source(store(v)(B), B) == store(v)(B).block_states[B].current_justified_checkpoint`.
   Therefore,

`store(v)(B).block_states[B].current_justified_checkpoint.epoch > store(v).justified_checkpoint.epoch`.
Observe that `on_block` executes `update_checkpoints(store, state.current_justified_checkpoint, state.finalized_checkpoint)` with `state.current_justified_checkpoint == store(v)(B).block_states[B].current_justified_checkpoint` which sets `store(v)(B).justified_checkpoint.epoch` to `max(store(v).justified_checkpoint.epoch,store(v)(B).block_states[B].current_justified_checkpoint)`.
This concludes the proof of the Lemma for this case.

Q.E.D.

Lemma 77.
For any validator `v`, let `store(v)(B)` be the value of `store(v)` after `v` receives block `B`, i.e. after it executes `on_block(store(v), B)` without raising exceptions.
If `get_voting_source(store(v)(B), B).epoch <= store(v).justified_checkpoint.epoch`, then `store(v).justified_checkpoint.epoch == store(v)(B).justified_checkpoint.epoch`.

Proof.
The Lemma can be proven by following the same reasoning outline in Lemma 76.

Q.E.D.

Lemma 80.
For any honest validator `v`, at any point in time, there exists at least a leaf block `B` in `store(v).blocks` such that `get_voting_source(store(v), B) == store(v).justified_checkpoint`

Proof.
The proof is by induction.

Base Case. At genesis time.
The store at genesis time corresponds to the execution of `get_forkchoice_store(genesis_state, genesis_block)`.
By the definition of `get_forkchoice_store` follows that the Lemma holds for the base case.

Inductive Case. Let `store(v)(e)` be the value of `store(v)` after executing the handler for event `e`.
We assume that there exists a block B in `store(v).blocks` such that

`get_voting_source(store(v), B) == store(v).justified_checkpoint` and prove that there exists a block `B’’` such that `get_voting_source(store(v)(e), B’’) == store(v)(e).justified_checkpoint`..

We now proceed by cases on the type of event `e`.

1. `on_attester_slashing` and `on_attestation`
   In this case, `store(v)(e)` satisfies the Lemma as none of these handlers alter any of the Store fields involved in the Lemma’s statement.
2. `on_block(store(v), B’)`.
   We distinguish between two sub-cases.
   a. `get_voting_source(store(v)(e), B’).epoch > store(v).justified_checkpoint.epoch`
      From Lemma 76 it follows that `store(v)(e).justified_checkpoint == get_voting_source(store(v)(e), B’)`.
      Therefore the Lemma is proved in this case with `B’’ == B’`.

   b. `get_voting_source(store(v)(e), B’).epoch <= store(v).justified_checkpoint.epoch`
      By Lemma 77, we have that `store(v)(e).justified_checkpoint == store(v).justified_checkpoint`.
      Also, by the definition of `on_block`, any of the fields involved in computing the value of `get_voting_source(store(v), B)` is unaltered. Therefore, `get_voting_source(store(v), B) == get_voting_source(store(v)(e), B)`.
      Therefore, the Lemma is proved by this case with `B’’ == B`.

3. `on_tick(store(v), time)`
   Given that `on_tick` essentially corresponds to a sequential execution of various calls to `on_tick_per_slot`, we reduce the proof for this case to the proof that the single execution of `on_tick_per_slot` maintains the invariant stated in the Lemma.
   Then let e correspond to the execution of `on_tick_per_slot(, t’).`
   We distinguish between two cases
   a. `first_slot(t’)==true`
      We distinguish between three sub-cases.
      i. `store(v)(e).justified_checkpoint.epoch > store(v).justified_checkpoint.epoch`
         The only place where `justified_checkpoint` could have been updated is in the execution of `update_checkpoints(store, store.unrealized_justified_checkpoint, store.unrealized_finalized_checkpoint)`. which would set `store(v)(e).justified_checkpoint == store(v)(e).unrealized_justified_checkpoint`.
         From Lemma 75 it follows that there exists a block `B’’` such that

`store(v)(e).justified_checkpoint == store(v)(e).unrealized_justifications[B'']`.
Given that `first_slot(t')==true`, we must have that `epoch(B'') < epoch(t')` and therefore `get_voting_source(store(v)(e), B'') == store(v)(e).unrealized_justifications[B'']` which proves the Lemma for this case.

ii. `store(v)(e).justified_checkpoint.epoch == store(v).justified_checkpoint.epoch`
Given that `on_tick_per_slot` does not alter any of the fields involved in the execution of `get_voting_source`, the Lemma is proved in this case.

iii. `store(v)(e).justified_checkpoint.epoch < store(v).justified_checkpoint.epoch`.
Impossible due to Lemma 35.

b. `first_slot(t')==false`
In this case, we have that `store(v)(e).justified_checkpoint.epoch == store(v).justified_checkpoint.epoch`
Given that `on_tick_per_slot` does not alter any of the fields involved in the execution of `get_voting_source`, the Lemma is proved in this case.

Q.E.D.

Lemma 90.
For any honest validators `v` and leaf block `B` in `store(v).blocks`, `store(v).justified_checkpoint.epoch >= get_voting_source(store(v), B).epoch`.

Proof.
The proof is by induction.

Base Case.
The store at genesis time corresponds to the execution of `get_forkchoice_store(genesis_state, gensis_block)`.
By the definition of `get_forkchoice_store` follows that the Lemma holds for the base case.

Inductive Case. Let `store(v)(e)` be the value of `store(v)` after executing the handler for event `e`.
We assume that for any honest validators `v` and leaf block `B` in `store(v).blocks`, `store(v).justified_checkpoint.epoch >= get_voting_source(store(v), B)`, and prove that for any block `B` in `store(v)(e).blocks`, `store(v)(e).justified_checkpoint.epoch >= get_voting_source(store(v)(e), B)`

1. `on_attester_slashing` and `on_attestation`
   In this case, `store(v)(e)` satisfies the Lemma as none of these handlers alter any of the Store fields involved in the Lemma's statement.
2. `on_block(store(v), B')`.
   Let B be any leaf block in `store(v)(e).blocks`.
   We consider two sub-cases
   a. `B` is already in `store(v).blocks`
      No line of code executed by `on_block(store(v), B')` affects the result of get_voting_source(store(v), B)..
      This, together with Lemma 35, implies the Lemma for this case.
   b. `B` is not in `store(v).blocks`
      In this case, `B = B'`.
      By Lemma 76, if `get_voting_source(store(v)(e), B).epoch > store(v).justified_checkpoint.epoch`, then `get_voting_source(store(v), B) == store(v)(e).justified_checkpoint.epoch`.
      This concludes the proof for this case.

3. `on_tick(store(v), time)`
   Given that `on_tick` essentially corresponds to a sequential execution of various calls to `on_tick_per_slot`, we reduce the proof for this case to the proof that the single execution of `on_tick_per_slot` maintains the invariant stated in the Lemma.
   Then let e correspond to the execution of `on_tick_per_slot(, t')`.
   We distinguish between two cases
      `first_slot(t')==true`
      Let `B` be any leaf block in `store(v)(e).blocks`.
      Given that `first_slot(t')==true`, we must have that `epoch(B) < epoch(t)` and therefore `get_voting_source(store(v)(e), B) == store(v)(e).unrealized_justifications[B]`.
      Also, given that `first_slot(t')==true`, `update_checkpoints(store, store.unrealized_justified_checkpoint, store.unrealized_finalized_checkpoint)` is executed which sets `store(v)(e).justified_checkpoint.epoch >= store(v)(e).unrealized_justified_checkpoint`.
      Lemma 74 then concludes the proof for this case.
   a. `first_slot(t')==false`
      In this case, none of the Store's fields involved in the Lemma's statement are changed. Therefore, the Lemma still holds for `store(v)(e)`.

Q.E.D.

Lemma 100.
For any honest validator `v`,

1. `get_filtered_block_tree(store(v)) != {}`, and
2. any leaf block `B` in `get_filtered_block_tree(store(v))` satisfies at least one of the two following conditions
   a. `get_voting_source(store(v), B).epoch == store(v).justified_checkpoint.epoch`
      or
   b.
```
(
  B in store(v).unrealized_justifications and
  store(v).unrealized_justifications[B].epoch >=
store(v).justified_checkpoint.epoch and
  get_voting_source(store(v), B).epoch + 2 >=
store(v).current_epoch
)
```

Proof.
Let us prove the two conditions separately.

- Condition 1.
  From Lemma 80, it follows that there exists at least one leaf block `B` that satisfies the following filtering condition inside `get_filtered_block_tree`:
  `get_voting_source(store(v), B).epoch == store(v).justified_checkpoint.epoch`

  From Lemma 30 it follows that `B` also satisfies the following filtering condition
  `store.finalized_checkpoint.root == get_ancestor(store, block_root, finalized_slot)`

- Condition 2.
  Given the definition of `get_filtered_block_tree`, all we need to show is that
  `store(v).justified_checkpoint.epoch == GENESIS_EPOCH` implies
  `get_voting_source(store(v), B).epoch == store(v).justified_checkpoint.epoch`
  This follows from Lemma 90 and the fact that `get_voting_source(store(v), B).epoch >= GENESIS_EPOCH`.

Q.E.D.

Lemma 102.
For any block `B`,
`process_justification_and_finalization(post_state(B)).current_justifi`

`ed_checkpoint.epoch >= post_state(B).current_justified_checkpoint.epoch`

Proof.
Due to its definition, `process_justification_and_finalization` can only increase the epoch of `current_justified_checkpoint`.
Q.E.D.

Lemma 103.
For any blocks `Bp` and `B`, such that `Bp` is an ancestor of `B` and `epoch(Bp) < epoch(B)`, `post_state(B).current_justified_checkpoint.epoch >= process_justification_and_finalization(post_state(Bp)).current_justified_checkpoint.epoch`.

Proof.
Given that `B` is from an epoch higher than `Bp`, `post_state(B)` already accounts for all the justifications included in `Bp`.

Q.E.D.

Lemma 104.
For any block `B`, `store.unrealized_justifications[B].epoch >= store.block_states[B].current_justified_checkpoint.epoch`.

Proof.
Observe that `store.block_states[B]` corresponds to `post_state(B)` and `store.unrealized_justifications[B]` corresponds to `process_justification_and_finalization(post_state(B)).current_justified_checkpoint`.

Lemma 102 then implies the Lemma.

Q.E.D.

Lemma 105.
For any blocks `Bp` and `B`, such that `Bp` is an ancestor of `B`, `get_voting_source(store, Bp).epoch <= get_voting_source(store, B).epoch`.

Proof.
Let us proceed by cases.

1. `epoch(Bp) == epoch(B) == current_epoch`
   In this case, `get_voting_source` for `B` and `Bp` reduces to `post_state(B).current_justified_checkpoint` and

`post_state(Bp).current_justified_checkpoint` respectively.
Due to the way `post_state` is defined, we have that `post_state(B).current_justified_checkpoint.epoch >= post_state(Bp).current_justified_checkpoint.epoch` which concludes the proof for this case
2. `epoch(Bp) <= epoch(B) < current_epoch`
In this case, `get_voting_source` for `B` and `Bp` reduces to `process_justification_and_finalization(post_state(B)).current_justified_checkpoint` and `process_justification_and_finalization(post_state(Bp)).current_justified_checkpoint` respectively.
Due to the way `post_state` and `process_justification_and_finalization` are defined, we have that `process_justification_and_finalization(post_state(B)).current_justified_checkpoint.epoch >= process_justification_and_finalization(post_state(Bp)).current_justified_checkpoint.epoch` which concludes the proof for this case.
3. `epoch(Bp) < epoch(B) == current_epoch`
Due to Lemma 104 and the way `get_voting_source` is defined, it suffices to prove that `store.block_states[B].current_justified_checkpoint.epoch == post_state(B).current_justified_checkpoint.epoch >= store.unrealized_justifications[Bp].epoch == process_justification_and_finalization(post_state(Bp)).current_justified_checkpoint.epoch`.
Lemma 103 concludes the proof for this case.

Q.E.D.

Lemma 110.
For any validator `v`, let `store(v)(B)` be the value of `store(v)` after `v` receives block `B`, i.e. after it executes `on_block(store(v), B)` without raising exceptions.
If `B` is a child of `get_head(store(v))`, then `get_head(store(v)(B)) = B`.

Proof.
Let `Bp = get_head(store(v))`.
Assume that `B` is a child of `Bp`.

We first prove the following condition
a) `B` is included in `get_filtered_block_tree(store(v)(B))`
Let us proceed by cases.
1. `get_voting_source(store(v)(B), B).epoch > store(v).justified_checkpoint.epoch`
From Lemma 76 it follows that`get_voting_source(store(v)(B), B).epoch ==`

`store(v)(B).justified_checkpoint.epoch`. Condition a) follows directly from this.
2. `get_voting_source(store(v)(B), B).epoch <= store(v).justified_checkpoint.epoch`
   We distinguish two sub-cases based on the fact that `Bp` is a leaf block of `store(v)` and Lemma 100:
   a. `get_voting_source(store(v), Bp).epoch == store(v).justified_checkpoint.epoch`
      From Lemma 105, we have that `get_voting_source(store(v)(B), B).epoch >= get_voting_source(store(v)(B),Bp).epoch`. This implies that `get_voting_source(store(v)(B).epoch == store(v).justified_checkpoint` which, in turn, implies condition a).
   b. `get_voting_source(store(v), Bp).epoch >= current_epoch - 2`
      Given, that `on_block` does not change `current_epoch` and that, due to Lemma 105, `get_voting_source(store(v)(B), B).epoch >= get_voting_source(store(v)(B), Bp).epoch`, condition a) is true in this case.

Given that

- by assumption, `Bp` is the block with the highest weight between any of its (filtered) siblings
- `B` has no siblings as `Bp` was a leaf block in `store(v)`

it follow that `B = get_head(store(v)(B))`

Q.E.D.

Lemma 120.
For any honest validator `v`, let `store(v)(t)` be the value of the `store(v)` after executing `on_tick(store(v), t)`.
If `fist_slot(t) == true`, then for any leaf block `B` in `get_filtered_block_tree(store(v)(t))`, we have that

1. `get_voting_source(store(v), B).epoch == store(v)(t).justified_checkpoint.epoch`.
2. `store(v)(t).unrealized_justifications[B] == store(v)(t).unrealized_justified_checkpoint`

Proof.
Given that `fist_slot(t) == true`, `update_checkpoints(store, store.unrealized_justified_checkpoint, store.unrealized_finalized_checkpoint)` is executed which sets

`store.justified_checkpoint.epoch >= store.unrealized_justified_checkpoint.epoch`.

Due to Lemma 74, if `B` is in `get_filtered_block_tree(store(v)(t))`, then the condition 2.b of Lemma 100 can only be satisfied if `store(v).unrealized_justifications[B].epoch == store(v).justified_checkpoint.epoch` which is the same condition that satisfied condition 2.a of Lemma 100. This concludes the proof for condition 1 of the Lemma.

We have that `store(v)(t).unrealized_justifications[B].epoch == store(v)(t).unrealized_justified_checkpoint.epoch` due the following facts:

- `store.justified_checkpoint.epoch >= store.unrealized_justified_checkpoint.epoch`
- `store(v).unrealized_justifications[B].epoch == store(v).justified_checkpoint.epoch`
- Lemma 74

Lemma 30 then implies condition 2 of the Lemma.

Q.E.D.

Lemma 130.
Assume the following conditions

1. at the beginning of epoch `e` all honest validators are fork-choice equivalent
2. message are delivered in time 0 during epoch `e`
3. the clocks of all honest validators are perfectly synchronized since the beginning of epoch `e`
4. no dishonest message is sent in epoch `e`
5. no message for an epoch previous to `e` is ever received by any of the honest validator during epoch `e`

For any slot `s`, let `tp(s)` be the time when an honest node is supposed to propose in slot `s`, `ta(s)` the time when an honest validator is supposed to attest in slot `s`, `store(v)(t)` be the store of validator `v` at time `t` and `p(s)` be the proposer for slot `s`.
Let us enumerate the slots by assigning 0 to the first slot of epoch `e`.
Let `P(s)` be defined as follows

- `P(-1) =` the result of any honest validator v executing `get_head(store(v)(tp(0)))`
- if the proposer `p(s)` of slot `s` is honest then `P(s)` corresponds to the block proposed by p(s) in slot s.
- if the proposer `p(s)` of slot `s` is dishonest, then `P(s) = P(s-1)`

The following conditions hold for any slot `s`

1. honest validators are fork-choice equivalent at time `tp(s)`
2. honest validators are fork-choice equivalent at time `ta(s)`
3. `P(s)` is a descendent of `P(s-1)`
4. At time `ta(s)`, honest validator attests for block `P(s)`
5. For any honest validator `v`, `get_voting_source(store(v)(ta(s)), P(s)).epoch == store(v)(ta(s)).justified_checkpoint.epoch`
6. For any honest validator `v`, `store(v)(ta(s)).unrealized_justified_checkpoint == store(v)(ta(s)).unrealized_justifications[P(s)]`

Proof.

The proof is by induction.

Base Case. Slot 0.
Let us look at each condition separately.

1. Condition 1.
   Follows directly from the Lemma's assumptions.
2. Condition 2.
   By time `ta(s)`, all honest validators have received the block proposed by `p(s)`, if `p(s)` is honest, or no message otherwise. Therefore, they are fork-choice equivalent.
3. Condition 3.
   Let us consider two cases.
   a. `p(s)` is honest
      The block proposed by p(s) is a child of `get_head(store(v)(tp(0))) == P(-1)`. Hence, condition 3 holds in this case.
   b. `p(s)` is dishonest
      By definition, we have that `P(0) = P(-1)`. Hence, condition 3 holds in this case.
4. Condition 4.
   Let us consider two cases.
   a. `p(s)` is honest
      By time `ta(s)`, all honest validators have received the block proposed by `p(s)`. Lemma 110 then implies this Condition.
   b. `p(s)` is dishonest.
      In this case, no block is sent and, therefore, `get_head(store(v)(tp(0)) = get_head(store(v)(ta(0))` which implies the Condition.
5. Condition 5.
   Implied by Lemma 120 condition 1, the definition of `P(s)` and the Lemma's assumption on fork-choice equivalence.
6. Condition 6.
   Observe that `P(-1)` corresponds to the result of `p(s)` executing `get_head` after executing `on_tick` in the first slot of epoch `e`.

Lemma 120 condition 2 therefore implies that
`store(v)(tp(0)).unrealized_justifications[P(-1)] ==`
`store(v)(tp(0)).unrealized_justified_checkpoint` for any honest validator `v`
As established by condition 3, `P(0)` is a descendant of `P(-1)`. Hence, when by time `ta(s)` any honest validator `v` has received `P(0)`, we have that
`store(v)(ta(0)).unrealized_justifications[P(0)].epoch >=`
`store(p(s))(ta(0)).unrealized_justifications[P(-1)].epoch`.
As a consequence of this, `compute_pulled_up_tip` (via `update_unrealized_checkpoints`) sets
`store(p(s))(ta(0)).unrealized_justifications[P(0)] ==`
`store(p(s))(ta(0)).unrealized_justified_checkpoint` which concludes the proof for this condition.

Inductive Case.
We assume that the Lemma holds for slot `s` and prove that it also holds for slot `s' = s + 1`
Let us look at each condition separately.

1. Condition 1.
   By inductive hypothesis, all validators are fork-choice equivalent at time `ta(s)`. By the Lemma's assumption, by time `tp(s')` they all receive the same set of messages (i.e. the attestations sent at time `ta(s)`). The Condition follows directly from this.
2. Condition 2.
   Can be concluded with a reasoning similar to the one outlined for Condition 2 of the Base Case.
3. Condition 3.
   Let us consider two cases.
   a. `p(s')` is honest.
      By inductive hypothesis, `get_head(store(p(s'))(ta(s))) = P(s)` and any message sent between `ta(s)` and `tp(s')` is an attestation message for `P(s)`. Hence, `get_head(store(p(s'))(tp(s'))) = P(s)`. Given that `p(s')` is honest, `P(s')` is a child of `P(s)` proving the Condition.
   b. `p(s')` is dishonest.
      By definition of `P`, `P(s') = P(s)` which proves the Condition.
4. Condition 4.
   Can be concluded with a reasoning similar to the one outlined for Condition 4 of the Base Case.
5. Condition 5.
   The inductive hypothesis and the conditions above imply that between time `ta(s)` and time `ta(s')`, honest validators receive at most block `P(s')` and attestations for block `P(s)`.
   Attestation messages do not alter any of the fields involved in the definition of Condition 5.

Due to the fact that `P(s')` is not from a previous epoch, `s'` is not the first slot of the epoch, we have that `get_voting_source(store(v)(ta(s')), P(s')) == get_voting_source(store(v)(ta(s')), P(s))`, which also implies that `store(v)(ta(s')).justified_checkpoint == store(v)(ta(s)).justified_checkpoint` which concludes the proof for this Condition.

6. Condition 6.
   Given that by condition 3, we have that `P(s')` is a descendant of `P(s)`, we have that `store(v)(ta(s')).unrealized_justifications[P(s')].epoch >= store(v)(ta(s)).unrealized_justifications[P(s)] == store(v)(ta(s')).unrealized_justifications[P(s)]`.
   Given that by inductive hypothesis `store(v)(ta(s)).unrealized_justified_checkpoint == store(v)(ta(s)).unrealized_justifications[P(s)]` and that the only block received between ta(s) and ta(s') is P(s'), due to the execution of `compute_pulled_up_tip` (via `update_unrealized_checkpoints`) we have that `store(p(s))(ta(s)).unrealized_justifications[P(s')] == store(p(s))(ta(s)).unrealized_justified_checkpoint` which concludes the proof for this condition.

Q.E.D.

Lemma 135.
For any honest validator `v`, `store(v).unrealized_justified_checkpoint.epoch >= store(v).justified_checkpoint.epoch`.

Proof.
The proof is by induction.

Base Case. At genesis time.
The store at genesis time corresponds to the execution of `get_forkchoice_store(genesis_state, genesis_block)`.
By the definition of `get_forkchoice_store` follows that the Lemma holds for the base case.

Inductive Case. Let `store(v)(e)` be the value of `store(v)` after executing the handler for event `e`.

We now proceed by cases on the type of event `e`.

1. `on_attester_slashing` and `on_attestation`
   In this case, `store(v)(e)` satisfies the Lemma as none of these handlers alter any of the Store fields involved in the Lemma's statement.

2. `on_block(store(v), B)`.
   In this case, we have that
   - `store(v)(e).justified_checkpoint.epoch == max(store(v).justified_checkpoint.epoch, post_state(B).current_justified_checkpoint)` and
   - `store(v)(e).unrealized_justified_checkpoint.epoch == max(store(v).unrealized_justified_checkpoint.epoch, process_justification_and_finalization(post_state(B)).current_justified_checkpoint)`.

   Let us consider the following two sub-cases:
   a. `store(v)(e).justified_checkpoint.epoch == store(v).justified_checkpoint.epoch`
      Given that `store(v).justified_checkpoint.epoch <= store(v).unrealized_justified_checkpoint.epoch <= store(v)(e).unrealized_justified_checkpoint.epoch`, the Lemma is proved in this sub-case.
   b. `store(v)(e).justified_checkpoint.epoch == post_state(B).current_justified_checkpoint`
      Given that `post_state(B).current_justified_checkpoint <= process_justification_and_finalization(post_state(B)).current_justified_checkpoint <= store(v)(e).unrealized_justified_checkpoint.epoch`, the Lemma is proved in this sub-case.

3. `on_tick(store(v), time)`
   Given that `on_tick` essentially corresponds to a sequential execution of various calls to `on_tick_per_slot`, we reduce the proof for this case to the proof that the single execution of `on_tick_per_slot` maintains the invariant stated in the Lemma.
   Then let e correspond to the execution of `on_tick_per_slot(, t').`
   We distinguish between two cases
   - `first_slot(t')==true`
     In this case, `store(v)(e).justified_checkpoint.epoch == max(store(v).justified_checkpoint.epoch, store(v).unrealized_justified_checkpoint.epoch)`.
     Let us consider two cases.
     We distinguish between three sub-cases.
     i. `store(v)(e).justified_checkpoint.epoch == store(v).justified_checkpoint.epoch`.
        Given that `store(v).justified_checkpoint.epoch <= store(v).unrealized_justified_checkpoint.epoch == store(v)(e).unrealized_justified_checkpoint.epoch`, the Lemma is proved in this case.

ii. `store(v)(e).justified_checkpoint.epoch == store(v).unrealized_justified_checkpoint.epoch`.
This case implies the Lemma directly.

- `first_slot(t')==false`
In this case, none of the Store's fields involved in the Lemma's statement are changed. Therefore, the Lemma still holds for `store(v)(e)`.

Q.E.D.

Lemma 140.
Assume the following conditions
1. at the beginning of epoch `e` all honest validators are fork-choice equivalent
2. message are delivered in time 0 during epoch `e`
3. the clocks of all honest validators are perfectly synchronized since the beginning of epoch `e`
4. no dishonest message is sent in epoch `e`.
5. no message for an epoch previous to `e` is ever received by any of the honest validator during epoch `e`

Let `tp(s)`, `ta(s)`, `p(s)` and `P(s)` be defined as per Lemma 130.
Let `store(v)(e+1)` be the store of an honest validator `v` after executing `on_tick(store(v), t)` with `epoch(t) == e+1` and `first_slot(t) == true`.
Then, the following condition holds:
`get_head(store(v)(e+1)) == P(SLOTS_PER_EPOCH)`.

Proof.
Let `t` be such that `epoch(t) == e+1` and `first_slot(t) == true`.
In alignment with the Lemma statement, we use `store(v)` to refer to the value of the Store of `v` before executing `on_tick`.
Observe that:
- Given that `epoch(P(SLOTS_PER_EPOCH)) < epoch(t)`, we have that `get_voting_source(store(v)(e+1), P(SLOTS_PER_EPOCH)) == store(v)(e+1).unrealized_justifications[P(SLOTS_PER_EPOCH)`].
- Given that `on_tick` does not alter `unrealized_justifications`, we have that `store(v)(e+1).unrealized_justifications[P(SLOTS_PER_EPOCH)] == store(v)(ta(SLOTS_PER_EPOCH)).unrealized_justifications[P(SLOTS_PER_EPOCH)]`.
- Given that no block is received since `ta(SLOTS_PER_EPOCH)`, we have that `store(v).unrealized_justified_checkpoint == store(v)(ta(SLOTS_PER_EPOCH)).unrealized_justified_checkpoint`.

Let us now prove that `P(SLOTS_PER_EPOCH)` is in `get_filtered_block_tree(store(v)(e+1))` by considering two cases:

1. `store(v)(e+1).unrealized_justifications[P(SLOTS_PER_EPOCH)] >= store(v).justified_checkpoint.epoch`
   Due to the fact that `update_checkpoints(store, store.unrealized_justified_checkpoint, store.unrealized_finalized_checkpoint)` is executed in `on_tick_per_slot`, we have that `store(v)(e+1).justified_checkpoint.epoch == max(store(v).justified_checkpoint, store(v).unrealized_justified_checkpoint)`.
   Lemma 135 then implies that `store(v)(e+1).justified_checkpoint.epoch == store(v).unrealized_justified_checkpoint`.
   Condition 6 of Lemma 130 implies that `store(v)(e+1).justified_checkpoint.epoch == store(v)(e+1).unrealized_justifications[P(SLOTS_PER_EPOCH)]`.
   Lemma 100 then concludes the proof for this case.
2. `store(v)(e+1).unrealized_justifications[P(SLOTS_PER_EPOCH)] < store(v).justified_checkpoint.epoch`
   This would imply `store(v)(e).unrealized_justifications[P(SLOTS_PER_EPOCH)] < store(v).justified_checkpoint.epoch`.
   Then condition 5 of Lemma 130 then implies that this case is impossible.
1.

Then the Lemma follows from the following conditions:

1. No additional branch is added to the set `get_filtered_block_tree(store(v)(e+1))` compared to the set `get_filtered_block_tree(store(v))` (as `on_tick(store(v), t)` executes `pull_up_tip` only on block `P(SLOTS_PER_EPOCH)`
2. `P(SLOTS_PER_EPOCH)` is the block with the highest weight amongst its (filtered) siblings at time `ta(SLOTS_PER_EPOCH)`
3. Between time `ta(SLOTS_PER_EPOCH)` and the beginning of epoch `e+1`, the only messages sent are attestations for block `P(SLOTS_PER_EPOCH)`.

Q.E.D.

Lemma 150.

Assume the following conditions

1. at the beginning of epoch `e` all honest validators are fork-choice equivalent
2. message are delivered in time 0 in epoch `e` to epoch `ef` (with `ef >  e`)

3. the clocks of all honest validators are perfectly synchronized since the beginning of epoch `e`
4. no dishonest message is sent in epoch `e` to epoch `ef`
5. no message for an epoch previous to `e` is ever received by any of the honest validator during epoch `e` up until at least epoch `ef`

Let `tp(s)`, `ta(s)`, `p(s)` and `P(s`) be defined as per Lemma 130.
The following conditions hold for any slot `s` in epochs `[e, ef]`

1. honest validators are fork-choice equivalent at time `tp(s)`
2. honest validators are fork-choice equivalent at time `ta(s)`
3. `P(s)` is a descendent of `P(s-1)`
4. At time `ta(s)`, honest validators attests for block `P(s)`
5. For any honest validator `v`, `get_voting_source(store(v)(ta(s)), P(s)).epoch == store(v)(ta(s)).justified_checkpoint.epoch`
6. For any honest validator v, store(v)(ta(s)).unrealized_justified_checkpoint == store(v)(ta(s)).unrealized_justifications[P(s)]

Proof.
The proof is by induction.

Base Case. The epoch `e` which corresponds to the range `[e, e]`.
The base case follows from Lemma 130.

Inductive Case.
We assume that the Lemma holds for any epoch in the range `[e, e']` and prove that it also holds for any epoch in the range `[e, e'']` with `e'' = e' + 1`.
Let `s'` be the last slot of epoch `e'` and `s''` be the first slot of epoch `e''`.
Lemma 130 implies that to prove the above it suffices to prove the following two conditions::

1. At the beginning of epoch `e''`, all honest validators are fork-choice equivalent
2. `P(s'')` is a descendent of `P(s')`.

Let us prove each condition separately.

1. Condition 1.
   From the inductive hypothesis, we know that at time `ta(s')`, all honest validators are fork-choice equivalent. Given the Lemma's assumption, by the end of slot `s'` they will have received the same set of messages and therefore they are fork-choice equivalent. A reasoning similar to the one used in Lemma 50 can be used to show that the execution of `on_tick` at the beginning of slot `s''` preserves fork-choice equivalence.
2. Condition 2.
   Let us consider two cases.
   a. The proposer `p(s'')` is honest.
      In this case, from Lemma 140 follows that `p(s'')` proposes a block with parent `P(s')`.

b. The proposer `p(s'')` is dishonest.
   No block is proposed in slot `s''` and therefore `P(s'') = P(s')` which implies the Condition.

The two conditions above and Lemma 130 then imply the Lemma.

Q.E.D.

Lemma 155.
Honest validators never commit slashable violations.

Proof.
The proof of this Lemma can be found here.
Q.E.D.

Lemma 160.
Let `T` be any finite value. Let network and nodes be asynchronous up to time `T` and let `S` be any possible state that a distributed system running under the assumptions listed at beginning of this document may end up at at time `T`.
There exists a possible sequence of events starting from state `S` that does not involve any dishonest node and that leads to justifying a checkpoint for an epoch `<= epoch(T) + 2`.

~~scheduling~~
Proof.
Let `M` be the set of all messages sent by time `T`.

Let `E` be any sequence of events induced by the following constraints:
1. At time `T`, all honest validators receive any of the messages in the set `M` that they have not yet received.
2. From time `T` onwards
   a. dishonest validators do not send any message.
   b. messages are delivered in time 0
   c. honest validators clocks are perfectly synced

We now show that the sequence of events `E` leads to justifying a checkpoint for an epoch `<= epoch(T) + 2`.

Lemma 40 and Lemma 50 imply that honest validators are fork-choice equivalent at the beginning of epoch `epoch(T)+2`.
Let `B' = get_head(store(v))` at the beginning of epoch `epoch(T)+2` for any honest validator `v`. Given that honest validators are fork-choice equivalent, the value of `B'` is uniquely determined.

According to Lemma 150, any block proposed from epoch `epoch(T)+2` onwards is for a descendent of `B'`.
Let `p'` be the proposer for the first slot of epoch `epoch(T)+2`.
Let `C'` be the checkpoint in epoch `epoch(T)+2`.
If `p'` is honest, then `C'` corresponds to the block proposed by `p'`. Otherwise, `C'` corresponds to `B'`.
According to Lemma 150, any attestation cast from epoch `epoch(T)+2` onwards is for a descendent of `C'`.
Lemma 155 implies that none of these attestations are slashable.
Hence, by the end of epoch `epoch(T)+2`, enough attestations to justify `C'` have been sent.

Q.E.D.

Lemma 170.
Let `T` be any finite value. Let network and nodes be asynchronous up to time `T` and let `S` be any possible state that a distributed system running under the assumptions listed at beginning of this document may end up at at time `T`.
Let `eh` be the first epoch `> epoch(T)+2` such that the proposer of its first slot is honest.
There exists an admissible sequence of events starting from state `S` that does not involve any dishonest node and that leads to finalizing a new checkpoint for an epoch `<= eh-1`

Proof.
Let `M` be the set of all messages sent by time `T`.

Let `E` be any sequence of events induced by the following constraints:

1. At time `T`, all honest validators receive any of the messages in the set `M` that they have not yet received.
2. From time `T` onwards
    a. dishonest validators do not send any message.
    b. messages are delivered in time 0
    c. honest validators clocks are perfectly synced

We now show that the sequence of events `E` leads to finalizing a new checkpoint for an epoch `<= eh-1`.

From Lemma 160, we know that an epoch `e' <= epoch(T)+2` will be justified.
Let us consider two cases.

1. The block proposed by the last honest proposer of epoch `e'` includes enough attestations to justify epoch `e'`.
   After executing `on_tick` at the beginning of epoch `e'+1`, all honest validators will therefore set `store(v).justified_checkpoint` to checkpoint `C'` (see Lemma 160 for the definition of `C'`).

Any attestation cast in epoch `e'+1` will therefore have `C'` as the source.
Let `B'' = get_head(store(v))` at the beginning of epoch `e'+1` for any honest validator `v`. Given that honest validators are fork-choice equivalent, the value of `B''` is uniquely determined.
Let `C''` be the checkpoint for epoch `e'+1`.
If the proposer `p''` of the first block of epoch `e'+1` is honest, then `C''` corresponds to the block proposed by `p''`. Otherwise, `C''` corresponds to `B''`.
According to Lemma 150, any attestation cast from epoch `e'+1` onwards is for a descendent of `C''`.
Hence, by the end of epoch `e'+1`, enough attestations to justify `C''` have been sent.
This, in turn, finalizes checkpoint `C'` in epoch `e'` which, by definition, is `<= eh -1`

2. The block proposed by the last honest proposer of epoch `e'` does not include enough attestations to justify epoch `e'`.
   Let `ph` be the proposer of the first slot of epoch `eh`, which, by assumption is honest.
   By following a reasoning similar to the one of Lemma 160, one can prove that epoch `eh-1` will be justified.
   `ph` includes any vote for epoch `eh-1`. Therefore, after honest validators receive the block proposed by `ph`, they set `store(v).justified_checkpoint` to a checkpoint for epoch `eh-1`.
   Following a reasoning similar to the one for the case above, one can prove that epoch `eh` will be justified which in turns finalizes epoch `eh-1`.

Q.E.D.

Corollary 180.
A distributed system running under the assumptions listed at the beginning of this document can never reach a state where it is impossible to finalize a new epoch.

Proof.
Follows from Lemma 170.

Q.E.D.

Acknowledgements

We thank Mikhail Kalinin and Alex Vlasov from ConsenSys, Aditya Asgaonkar, Francesco D'Amato and Luca Zanolini from the Ethereum Foundation for their reviews and insightful comments.