F3: GossiPBFT[1] implementation of Fast Finality in Filecoin (v2)

Design document

[1] Formerly known as Granite.

Introduction

GossiPBFT design choices

Implementing Byzantine fault-tolerant (BFT) protocols is notoriously error-prone and sometimes these protocols are overly complex. GossiPBFT implementation of F3 focuses on ***design simplicity*** and ***low latency***. Besides, some of our design choices for GossiPBFT stem from the specific setting in which F3 implementation will be deployed, namely:

- Relatively large population of about 3500 consensus participants (Filecoin SPs), with the goal of supporting 10x more participants easily and, in the future, 100x+ more with possible tradeoffs (e.g., using committee selection and/or graceful degradation of deciding less frequently).
- Existence of Expected Consensus (EC) as the consensus engine for Filecoin, which, despite no finality (EC favors availability in the CAP theorem design space), effectively powers a $2B USD market cap L1, with embedded cryptoeconomic incentives for Filecoin participants. Changing these dramatically would be arguably very challenging and involve a lot of uncertainty in the Filecoin community.

The constraints of this deployment setting influence GossiPBFT design choices.

- Since EC already uses a broadcast channel, implemented by GossipSub for communication among consensus participants, and given the sheer number of participants, GossiPBFT also uses a ***broadcast*** **communication** channel.
- Besides, protocol robustness and low latency requirements go against leader-based protocols which anyway have difficulties scaling to a large number of participants. To this end, GossiPBFT is a ***leaderless*** protocol in which all participants execute the same code in every round without obtaining particular temporary roles (e.g., a leader).

Experience in BFT consensus protocol design shows that it is better to rely on a proven protocol than to invent a new one. However, strong reasons and the requirement for interoperability with EC drove us to **customize GossiPBFT to EC and Filecoin**. Still, GossiPBFT is not entirely novel. It is a variant of the celebrated and seminal Castro/Liskov PBFT protocol and in particular its original version which uses evidence attached to messages. That said, GossiPBFT improves on many problematic design choices of PBFT, notably: leader bottleneck and DoS surface and complex view change protocol (see this section for details).

The GossiPBFT protocol and correctness proof assume an adversary which can control up to 33% of total quality-adjusted power (QAP, participants' weight) in EC and tolerates partially synchronous communication. In partially synchronous communication, honest participants might be prevented for arbitrarily long periods of time from communicating among each other, yet there is an assumption that they will eventually be able to communicate. While there are consensus protocols which tolerate an adversary of up to 50% of QAP, they rely on strong synchrony assumptions for correctness, i.e., even if only 1 honest participant is partitioned from the rest, Agreement can be violated (through equivocation of Byzantine participants). The assumption of synchrony for correctness on relatively small time scales (which we intuitively require for providing "Fast Finality") with a large number of participants which communicate over a gossip network, is a dangerous one, which we prefer not to make.

In the following, we first provide a brief, informal overview of GossiPBFT, which is followed by a detailed model, pseudocode and a correctness proof. In Appendix A we discuss and give a version of the GossiPBFT protocol (now deprecated) which models GossipSub as reliable-broadcast (instead of best-effort broadcast) and that uses this to forgo direct evidence piggybacking to protocol messages.

GossiPBFT protocol overview

While GossiPBFT is in some sense novel and tuned to EC/Filecoin and specific requirements of F3, it maintains key invariant structures and the common-case message pattern of the seminal PBFT protocol (Castro and Liskov, OSDI99). GossiPBFT is round-based, with each round consisting of 3 phases:

- QUALITY(in round=0)/CONVERGE(in round>0) phase, resembling PBFT PRE-PREPARE phase
- PREPARE phase (resembling that of PBFT)
- COMMIT phase (resembling that of PBFT)

Despite a similar structure, GossiPBFT avoids some pitfalls of PBFT including:

- PBFT rotating leader approach, which may give surface to Denial of Service attacks and can slow down the protocol overall, impacting also its scalability. Instead, GossiPBFT relies on quasi-leader election based on a verifiable random function (VRF) to break symmetry (polyvalence) of problematic executions in the CONVERGE step. While this sacrifices deterministic guarantees of PBFT and provides probabilistic guarantees, we believe this tradeoff is worthwhile.
- Complex and error-prone PBFT view change protocol. GossiPBFT embeds timeouts and disagreements into an explicit vote for a special value $\perp$ and relies on piggybacked evidences (compacted aggregate multisigs) to replace the view-change functionality and embed it into the common case 3-phase protocol.

Moreover, while all-to-all communication in PREPARE and COMMIT steps of PBFT is sometimes cited as its weakness, in our case, since Filecoin/EC use GossipSub already, this design decision is very appropriate. In fact, even the first step of GossiPBFT (QUALITY/CONVERGE) uses all-to-all communication to make the protocol more robust and to turn a leader-based protocol such as PBFT into a quasi-leaderless protocol.

In addition, GossiPBFT in its base version, which uses best-effort broadcast, embeds evidence (effectively participant multisigs) in messages, much like the original OSDI'99 PBFT. While this was until recently considered expensive (albeit elegant and simple), efficient BLS and Schnorr signature compaction and aggregation allow us to revisit this design choice.

The result is a streamlined protocol, with the classical three-phase message pattern repeating over rounds, which is relatively simple to reason about and implement. We also provide a version of GossiPBFT without evidence piggybacking, which in turn relies on a reliable-broadcast communication primitive and more elaborate background message validation.

A special highlight of GossiPBFT protocol is its QUALITY phase which practically mandates a pre-agreement on a finalized chain prefix across a 2/3rd majority of QAP, to help with ECcompatibility and to make Filecoin power leakage attacks with 33%+ QAP adversary considerably more difficult to mount (as detailed in Appendix B). Due to EC compatibility requirements, a participant in GossiPBFT never votes (sends a protocol message) on a value it does not locally have, unless it already observes a 2/3rd majority of participants voting for that value.

Preliminaries

Chains, instances, weight and randomness

Each value the protocol deals with (proposal, value, decision) is a chain (list of tipsets) read from participant's local instance of Expected Consensus (EC) and input to GossiPBFT. As such, every chain has a weight and a power table. Thus, each value also defines participants' powers (see definitions of power in the next section).

F3 creates multiple (infinitely many) instances of the GossiPBFT protocol. In each instance, GossiPBFT *finalizes* (we also say *decides*) one chain and outputs a Proof of Finality (PoF).

baseChain is the chain defining the "starting point" and participant membership for an instance of GossiPBFT. Each GossiPBFT instance is parametrized with a baseChain. A GossiPBFT instance considers it a globally known, minimal prefix that every chain proposed to that instance must include. All chains not prefixed by baseChain are ignored by the protocol.

In practice, when F3 creates a new GossiPBFT instance, it uses the most recently finalized chain that has been output by the previous GossiPBFT instance as baseChain for the new instance. **Thus, a new instance only starts once the previous instance has been terminated with a decision.** This makes it possible for participants to calculate the power table for new instances in advance, and to build on the previously terminated output. In particular, a new instance only starts 2 epochs after the epoch timestamp of the latest finalized tipset (or immediately if that epoch has already started). On start of a new f3 instance, the participant's proposal as input chain is its local ECChain, pruned so that the head epoch is at most that of the second latest EC epoch (in order to increase the chance of pre-agreement on the input chain, and reaching decision in the first round of GossiPBFT). The power table used for the a new f3 instance is that of the state of the finalized chain 10 f3 instances in the past. That is, let the latest finalized epoch *e* at instance *i*, the current epoch *e'*, then the new f3 instance *i+1* starts once e'>= e+2, with an input proposal such that proposal.head().epoch <= e'-1, and a the power table resulting from the state of the finalized chain on termination of instance *i-9.* The first GossiPBFT instance uses a default hard-coded baseChain. Messages exchanged between participants in one GossiPBFT instance are always annotated with a unique identifier of that instance and the implementation ignores all other messages. In the following, all descriptions of GossiPBFT itself concern a *single* instance. We omit instance identifiers for better readability.

The **weight** of a chain c is the weight of c's head tipset. The weight of a tipset t is the number of EC tickets in t plus the weight of t's parent tipset. Each block in t contains at least one ticket, but it can contain multiple ones if the proposer of the block was selected multiple times as proposer in the same EC epoch.

Randomness can be extracted from each chain, denoted ***Randomness(chain)***. In practice, *Randomness(chain)* returns the last beacon entry common to all tipsets forming the head of the *chain*.

Power

Quality-adjusted power (in short, **power** or **QAP**) is defined for each participant in a power table. The power table is a function of a chain, and thus the participants' power always needs to be defined with respect to a particular chain. The power table of the baseChain defines the membership (including participants' QAP) of the GossiPBFT instance.

Let *p* be a participant.

Power(p) ∈ ***[0, 1]*** is the relative QAP of *p* in expected consensus defined by the power table that corresponds to *baseChain* (denoted by **PT(baseChain),** or simply **PT**). *Power(p)* is expressed as a fraction of total QAP of all participants in the PT.

We define the power of a set of participants P as the sum of their individual powers.

$$Power(P) \;=\; \sum_{p \in P} Power(p)$$

Finally, let M be a set of messages from distinct senders. Then, we define Power(M) to be the sum of all participant powers across a union of message senders, i.e.,

$$Power(M) \;=\; \sum_{m \in M} Power(m.sender)$$

Adversary

We assume an adversary that

- Can corrupt participants with a total QAP of at most ⅓, making them Byzantine and coordinating them.

- Can delay messages among non-Byzantine participants for limited periods of time such that a partially synchronous model applies (see below).
- Cannot break the cryptographic primitives we use in the protocol.

Besides, we assume that all non-Byzantine participants are *rational* and that they will:

- Never send a message for any value that their EC instance locally has not already delivered. To see why this is intuitively rational, note that if GossiPBFT finalizes a tipset (chain) which EC does not locally have, a participant could not immediately mine in Filecoin/EC and receive cryptoeconomic awards for mining.
- Otherwise follow the protocol and profit from the value GossiPBFT/F3 and their faster finality bring to EC and the Filecoin ecosystem.

In the following, when we sometimes say *honest*, we mean *rational* or non-Byzantine participant.

Network model

Best-effort broadcast/Gossipsub

Our protocol implementation will likely use GossipSub (like Filecoin EC) to disseminate protocol messages, implementing a broadcast channel among participants. We assume and model GossipSub to satisfy the following properties of the *best-effort broadcast primitive* (denoted BEBroadcast hereafter):

- If p and p' are honest, then every message broadcast by p is eventually delivered by p'.
- No message is delivered unless it was broadcast by its sender.

Partially synchronous model and Δ-Synchrony

We say the system is Δ-synchronous if the combined computation and communication delay between any two honest participants is smaller than Δ, which is known to nodes. We assume that, under Δ-synchrony, any message broadcasted by an honest participant is delivered by every honest participant within a time bound of Δ.

In practice, if GossipSub is used for BEBroadcast, Δ is effectively the assumed upper bound on GossipSub latency (e.g., a few seconds).

For Termination, we assume a classical partial (eventually) synchronous model in which after GST (Global Stabilization Time), there is an unknown bound on communication delays among non-Byzantine nodes. To this end, we assume that the algorithm's estimate of Δ, used for protocol timeouts, doubles over rounds of the GossiPBFT protocol.

Message format, signatures and equivocation

Messages have the following fields *<sender, signature, msgType, value, [rnd, evidence, ticket]>*, where rnd, evidence and ticket are optional fields. We refer to a *field* of message m, with m.*field*.

All messages broadcast by a participant have their participant ID in the sender field and contain a digital signature by that participant (m.signature) over (msgType || value || rnd), as well as the GossiPBFT instance identifier that we omit for readability. The protocol assumes aggregatable signatures (e.g., BLS, Schnorr), resilient to rogue public key attacks (see e.g., Boneh, Drijvers and Neven construction).

The receiver of a message only considers messages with valid signatures and discards all other messages as invalid. We sometimes omit the sender IDs and signatures in further descriptions for better readability and where obvious.

Two (or more) messages m1 and m2 are called *equivocating messages*, if m1.sender=m2.sender AND m1.value≠m2.value AND m1.msgType=m2.msgType AND (if applicable) m1.rnd=m2.rnd. We call m1.sender an *equivocating sender*.

A set of messages M that does not contain equivocating messages is called *clean*.

In steps of the GossiPBFT protocol where an honest participant waits for a clean set of messages M, the protocol is correct regardless of whether an implementation

- immediately filters out from M **all** messages from equivocating senders as they are received, or
- leaves at most one message from an equivocating sender in M.

We recommend an implementation to apply the former approach

F3 Properties (recalling from F3 High-level Design)

Agreement. If two honest participants finalize (*i*,*h,**) and (*i*,*h*',*), then *h* = *h*'.

Validity. If an honest participant finalizes (i,h,*), then *h is a prefix of the* canonical_p' input chain of some honest participant p' in instance i.

Proof of Finality. If an honest participant finalizes (i,h,PoF), then PoF is a signature of 2/3rd QAP majority corresponding to PT(baseChain) input in instance i of some honest participant.

Progress. If the system is Δ-synchronous, i.e.:

- All honest participants can communicate within a known time bound Δ, and
- All honest participants invoke F3(i, *, *) at most Δ apart from each other,

Let c be the heaviest common prefix of the inputs of all honest participants in instance i. Then, if an honest participant finalizes (*i'*,*c'*,*), then *c is a prefix of c'* with probability > ⅔.

Termination. Every call to F3 eventually returns with probability 1.

EC compatibility. An honest node never contributes to forming a 2/3rd QAP (super-majority) of votes for a chain c which its local EC instance did not locally already deliver, unless it already observes evidence of such a super-majority for chain c.

GossiPBFT implementation of F3

Predicates and functions used in the protocol

- **isPrefix(a,b)**
  - returns TRUE iff a is a prefix of b. (Each chain is also a prefix of itself.)
- **StrongQuorum(prefix,M)**
  - Where M is a clean set of messages of the same type and the same round
  - Let P be the set of participants who are senders of messages in M such that their message contains a value with prefix as a prefix. More precisely:
    Let *P={p ∈ PT : ∃ m∈ M: m.sender=p AND isPrefix(prefix,m.value)}*
  - then the predicate returns TRUE iff Power(P)>⅔
- **HasStrongQuorumValue(M)**
  - Where M is a clean set of messages of the same type and the same round
  - The predicate returns TRUE iff there is a value v, such that there is a set P of participants who are senders of messages in M such that their message value is exactly v and Power(P)>2/3. More precisely:
  - *HasStrongQuorumValue(M) = ∃ v:*
    Power({p ∈ PT : ∃ m∈ M: m.sender=p AND m.value=v})>⅔
- ***StrongQuorumValue(M)***
  - If **HasStrongQuorumValue(M)** holds, returns *v*, nil otherwise.
- **GreatestTicketProposal(M)**
  - Let *M* be a clean set of CONVERGE messages for the same round.
  - The predicate returns m.value and m.evidence, such that *m* is the message in *M* with the greatest ticket after adjusting for the power of the sender (m.ticket*m.sender.power). M will never be empty as the participant must at least deliver its own CONVERGE message.
- **Aggregate(M)**
  - Where M is a clean set of messages of the same type T and round r, with v=StrongQuorumValue(M)≠nil
  - Let M' ← {m ∈ M : m.value = StrongQuorumValue(M)} \* filter M
  - returns a tuple <T, r, v, participants, agg-sig> where participants are all participants such that m.sender∈ *M' (*in some compact/compressed

representation, e.g., a bitmask with optional run-length encoding) and agg-sig is aggregate signature (BLS or Schnorr) across all of those participants on T||r||v.

GossiPBFT pseudocode (main algorithm)

See also PlusCal/TLA+ specification here (to be updated)

F3(inputChain, baseChain) returns (chain, PoF):

```
1:  round ← 0;
2: decideSent ← False;
3: proposal ← inputChain;  \* holds what participant locally believes should be a decision
4: timeout ← $2*\Delta$
5: value ← proposal     \* used to communicate voted value to others (proposal or $\bot$)
6: evidence ← nil       \* used to communicate optional evidence for voted value
7: C ← {baseChain}

8: while (not(decideSent))  {
9:      If (round = 0)
10:             BEBroadcast <QUALITY, value, instance>; trigger (timeout)
11:             collect a clean set M of valid QUALITY messages from this instance
                until StrongQuorum(proposal, M) OR timeout expires
12:             Let C ← C $\cup$ {prefix: isPrefix(prefix,proposal) AND StrongQuorum(prefix,M)}
13:             proposal ← heaviest prefix $\in$ C \* this becomes baseChain or sth heavier
14:             value ← proposal

15:     If (round > 0):                                              \* CONVERGE
16:             ticket ← VRF(Randomness(baseChain) || instance || round)
17:             value ← proposal        \* set local proposal as value in CONVERGE message
18:             BEBroadcast <CONVERGE, value, instance, round, evidence, ticket>; trigger(timeout)
19:             collect clean set M of valid CONVERGE msgs from this round and instance
                until timeout expires
20:             prepareReadyToSend ← False
21:             while (!prepareReadyToSend){
22:                     value, evidence ← LowestTicketProposal(M)  \* leader election
23:                     if (evidence is a strong quorum of PREPAREs AND mightHaveBeenDecided(value, r-1)):
24:                             C ← C $\cup$ {value}
25:                     If (value $\in$ C)
26:                             proposal ←value \* we sway proposal if the value is EC compatible (i.e., in C)
27:                             prepareReadyToSend ← True    \* Exit loop
28:                     Else
29:                             M = {m ∈ M | m.value != value AND m.evidence.value != evidence.value} \* Update M for next iteration }

30:     BEBroadcast <PREPARE, value, instance, round, evidence>; **trigger**(timeout) \* evidence is nil in round=0
31:     **collect** a clean set M of valid PREPARE messages from this round and instance
        **until** (HasStrongQuorumValue(M) AND StrongQuorumValue(M) = proposal)
                OR (timeout expires AND Power(M)>2/3)
32:     If (HasStrongQuorumValue AND StrongQuorumValue(M) = proposal) \* strong quorum of PREPAREs for local proposal

33:             value ← proposal                \* vote for deciding proposal (COMMIT)
34:             evidence ← Aggregate(M)         \* strong quorum of PREPAREs is evidence
35:     Else
36:             value ← ⊥               \* vote for not deciding in this round
37:             evidence ← nil

38:     BEBroadcast <COMMIT, value, instance, round, evidence>; **trigger**(timeout)

39:     **collect** a clean set M of valid COMMIT messages from this round and instance
        **until**     (HasStrongQuorumValue(M) AND StrongQuorumValue(M) ≠ ⊥)
                OR (timeout expires AND Power(M)>2/3)
40:     If (HasStrongQuorumValue(M) AND StrongQuorumValue(M) ≠ ⊥)              \* decide
41:             BEBroadcast <DECIDE, instance, StrongQuorumValue(M), Aggregate(M))
42:             decideSent ←TRUE
43:     Else if ( ∃ m.value ≠ ⊥ s.t. mightHaveBeenDecided(m.value, r))        \* value was possibly decided by others
44:             C ← C ∪ {m.value}               \* add to candidate values if not there
45:             proposal ← m.value;             \* sway local proposal to possibly decided value
46:             evidence ← m.evidence           \* strong PREPARE quorum is evidence for the value (it exists because the value might have been decided)
47:     Else                                    \* no participant decided in this round
48:             evidence ← Aggregate(M) s.t.  Power(M)>⅔ AND ∀ m ∈ M, m.value = ⊥      \* strong quorum of COMMITs for ⊥ is evidence
49:     round ← round + 1;
50:     timeout ← timeout * 1.3}  \*end while
51: **collect** a clean set M of valid DECIDE messages \* Collect strong quorum of decide outside
     **until** (HasStrongQuorumValue(M))                \* the round loop
52:  return (StrongQuorumValue(M), Aggregate(M))
```

```
                                                                    \* decide anytime
53: upon reception of a valid <DECIDE, instance, v, evidence> AND not(decideSent)
54:     decideSent ← TRUE
55:     BEBroadcast <DECIDE, instance, v, evidence)
56:     go to line 51.
```

The helper function mightHaveBeenDecided returns False if, given the already delivered messages, the participant knows for a fact that no correct participant could have decided the given value in the given round, even in the presence of an adversary controlling <⅓ the QAP equivocating, and True otherwise:

```
57: mightHaveBeenDecided(value, round):
58:     M ← { m | m.step = COMMIT AND m.round = round AND m is valid}
59:     M' ← { m | m ∈ M AND m.value = value }
60:     return Power(M') + (1-Power(M)) >= ⅓
```

Observe that if mightHaveBeenDecided(value, round) returns False then there is a strong quorum of COMMITs for bottom, as we prove in Lemma Strong Quorum for ⊥. In fact, the result of mightHaveBeenDecided(value, round) is tight, it only returns True when the participant cannot be certain that no other non-Byzantine participant decided the value in the round as we show in Lemma 1. That is if Power(M)=1 then the participant can return True or False with certainty, but otherwise it does not know with certainty what the remainder of the not-yet-delivered COMMIT messages might contain as value and what subset of them was delivered by some other participant.

Round catch-up sub-protocol

We describe in a document titled GossiPBFT Message Rebroadcast an extension of the protocol outlined in this document in order to deal with message loss before GST and prevent DDOS attacks by spamming messages. The extension involves the usage of queues of sent and received messages, along with a rebroadcast of sent messages under some conditions, and a procedure to jump forward to a future round, skipping intermediate rounds, under some conditions. We refer to that document for the proofs of correctness under that model of partial synchrony with lossy channels, which builds upon the proofs of this document.

Valid messages and evidences

The *Valid* predicate (referred to in lines 11, 19, 31, and 39, 51, 53) is defined below.

```
Valid(m) returns bool:                                  | For a message m to be valid,

● If m is not properly signed:                          | m must be properly signed.
        return False
● If m.instance != instance                             | m must match the instance ID.
        return False
● If m.sender has zero power or is unknown              | The sender must have power and
  be known
        return False
● If NOT (m.value = ⊥ OR baseChain.isPrefix(m.value)) | m's value must extend baseChain
  or be ⊥
        Return False
● If m.step != CONVERGE AND m.ticket != nil             | ticket must be nil if not CONVERGE
        Return False

● If m.step = QUALITY AND                               |  A QUALITY message must
      (m.round != 0 OR m.value = ⊥)                     |  have round 0 and non-⊥ value.
      return False                                      |

● If m.step = CONVERGE AND                              | A CONVERGE message
    (m.ticket does not pass VRF validation              | must have a valid VRF ticket
     OR m.round = 0 OR m.value = ⊥)                     | and round > 0.
        return False

● If m.step = DECIDE AND                                |  A DECIDE message must
    (m.round != 0 OR m.value = ⊥)                       |  have round 0 and non-⊥ value.
    return False

● If m.step ∈ {QUALITY}  AND m.evidence != nil          | QUALITY do not require evidence
    return False

● If m.step ∈ {CONVERGE, PREPARE
    COMMIT, DECIDE} AND                                 | CONVERGE, PREPARE, COMMIT &
    NOT ValidEvidence(m)                                | DECIDE must contain valid evidence.
        return False

● return True
```

The *ValidEvidence* predicate is defined below (for other definitions used below, see Preliminaries). Note that QUALITY and PREPARE messages do not need evidence.

```
ValidEvidence(m =  <step, value, instance, round, evidence, ticket>):
---------------------------------------------------------------------------

If ( step = PREPARE and round = 0)                              | in the first round,
        AND (evidence = nil)                                          | evidence for PREPARE
                return True                                           | is nil

If (step = COMMIT and value = ⊥)                                      | a COMMIT for ⊥ carries no evidence
        AND (evidence = nil)
                return True

If (evidence.instance != instance)                                    | the instance of the evidence must be
the
                return False                                          | same as that of the message

                                                                      | valid evidences for
return  (step = CONVERGE OR (step = PREPARE AND round>0)              | CONVERGE and PREPARE in
        AND ( ∃ M: Power(M)>⅔ AND evidence=Aggregate(M)               | round>0 is strong quorum
        AND ((∀ m' ∈ M: m'.step = COMMIT AND m'.value = ⊥)                    | of COMMIT msgs for ⊥
                OR (∀ m' ∈ M: m'.step = PREPARE AND                   | or PREPARE msgs for
                        m'.value = value))                            | CONVERGE   value
        AND (∀ m' ∈ M: m'.round = round-1)))                          | from previous round


OR (step = COMMIT                                                     | valid COMMIT evidence
    AND ((∃ M: Power(M)>⅔ AND evidence=Aggregate(M)                   | is a strong quorum
                AND ∀ m' ∈ M: m'.step = PREPARE                       | of PREPARE messages
                AND ∀ m' ∈ M: m'.round = round                        | from the same round
                AND ∀ m' ∈ M: m'.value = value)                       | for the same value, or
            OR (value = ⊥))                                           | COMMIT is for ⊥ with
                                                                      | no evidence

OR (step = DECIDE                                                     | valid DECIDE evidence
    AND (∃ M: Power(M)>⅔ AND evidence=Aggregate(M)                    | is a strong quorum
        AND ∀ m' ∈ M: m'.step = COMMIT                                | of COMMIT messages
        AND ∀ m' ∈ M: m'.round = round                                | from the same round
        AND ∀ m' ∈ M: m'.value = value))                              | for the same value
---------------------------------------------------------------------------
```

Correctness proof

We omit the GossipBFT instance identifier everywhere for readability, as all the properties and their proofs only ever refer to the same instance.

Termination

Lemma. (Compatible Proposals) A participant never changes its proposal from inputChain to a chain that is not in C, other than baseChain.

Proof. The process changes the proposal in lines 13, 26, and 45**.** Line 13 selects the value directly from C, line 26 is only executed if the value set to proposal is in C (check in line 25), and line 45 is immediately preceded by line 44 that adds the value to C.

Lemma. (Strong Quorum for ⊥). If mightHaveBeenDecide(v, r) = False for some value v received in a valid COMMIT/CONVERGE message in lines 43, 23 , then (i) there is a strong quorum of COMMIT messages for ⊥ and (ii) no non-Byzantine participant decided in round r.

Proof. (Strong Quorum for ⊥). Assume by contradiction that v was decided by some non-Byzantine participant p in round r. As a result, p received a strong quorum of COMMIT messages for v in round r. By quorum intersection and the adversary controlling less than ⅓ QAP, at least a weak quorum of these messages were sent by non-Byzantine participant. Let us define M ← { m | m.step = COMMIT AND m.round = round } and M' ← { m | m ∈ M AND m.value = v}. Since at least a weak quorum of non-Byzantine participants sent COMMITs for v, then Power(M') > ⅓ . But if mightHaveBeenDecide(v, r) = False then, by line 60, Power(M') + (1-Power(M)) < ⅓. <-> 1-Power(M) < ⅓ - Power(M'), with 1>=Power(M)>⅔ by line 39. But if Power(M') > ⅓ then 1-Power(M) < ⅓ - Power(M') <-> Power(M) > 1, which is impossible.

Now assume by contradiction that there is no strong quorum of valid COMMIT messages for ⊥. This means either that Power(M') > ⅓ (previous case), or that there is another value w != v such that Power(M') + Power(M") > ⅓ where M"= { m | m ∈ M AND m.value = w}. As COMMIT messages for a value are justified by a strong quorum of PREPAREs for that value, both COMMITs for v and w must have a strong quorum of PREPAREs for them. By quorum intersection and the fact that the adversary controls less than a weak quorum, this reaches a contradiction where some non-Byzantine participants sent multiple PREPARE messages for v and for w in round r.

QED (Strong Quorum for ⊥)

Lemma. (Advancing Rounds) *If p is non-Byzantine and no non-Byzantine participant decides up to round r, then p advances to round r+1.*

Proof. (Advancing Rounds) The only statements in the algorithm that potentially block indefinitely are waiting for messages from participants with valid evidence holding more than ⅔ the QAP, or by at least one valid message that must have been received within the timeout in CONVERGE step. By construction, participants receive their own messages instantly before the timeout in the CONVERGE step, which contains a value in C allowing the process to exit the while loop by the Compatible Proposals and Strong Quorum for ⊥ Lemma. By assumption that non-Byzantine participants hold more than ⅔ of QAP, that no non-Byzantine participant decides up to round r and by best effort broadcast, every message broadcast by a non-Byzantine participant will eventually be delivered by every non-Byzantine participant (guaranteeing reception of messages from non-Byzantine participants amounting to more than 2/3rd QAP).

Since non-Byzantine participants only broadcast messages with valid evidence, p will eventually receive enough messages in each step of round r. Since, by assumption of the lemma, p does not decide in round r, p advances to r+1.

QED (Advancing Rounds)

Lemma. (Synchronous Round) *Under Δ-synchrony (after GST), eventually, if a non-Byzantine participant p enters round r > 0 at time t, then all honest participants enter round r by t+timeout-Δ. We call r a synchronous round.*

Proof. (Synchronous Round) Since after GST, all messages among honest nodes are delivered within Δ, and since round timeouts are increased by 1.3 (line 50), it is not difficult to see that there will be a round r, such that: 1) timeout is big enough so 2) no honest participant entering r at time t progresses beyond CONVERGE phase due to timeout in line 19, before 3) all honest participants receive all messages from all other honest participants from rounds earlier than r , process them and 4) enter r by the time *t+timeout*-Δ.

QED (Synchronous Rounds)

Lemma. (Consistent Initial Proposals) *If two honest participants set their proposal variables to the respective values v and v' in the QUALITY step, then isPrefix(v, v') OR isPrefix(v', v) at the end of the QUALITY step.*

Proof. (Consistent Initial Proposals) Follows directly from line 12 and definition of StrongQuorum predicate.

QED (Consistent Initial Proposals)

Lemma. (Consistent Proposals) *Let v and v' be the values of the respective proposal variables of two correct participants. Then isPrefix(v, v') OR isPrefix(v', v) holds forever after the QUALITY step.*

Proof. (Consistent Proposals) In short, this invariant (*isPrefix(v, v') OR isPrefix(v', v)*) is established by Consistent Initial Proposals Lemma and because each update to an honest participant's proposal maintains this invariant.

To prove this, note that an honest participant can update its proposal, after the end of the QUALITY step, only in line 26, and/or line 45.

Line 26 maintains the invariant due to the way the protocol initializes (line 12) and updates C. That is, only new values added to C are those that gather support from a strong quorum of PREPAREs (lines 24, 44), a correct participant only sends a PREPARE for a value that was a

prefix of a strong quorum of values proposed in QUALITY in the first place, or a value for which it saw a strong quorum of PREPAREs. All values that are prefixes of a strong quorum of values in QUALITY are prefixes of each other by quorum intersection, as proven by Consistent Initial Proposals.
Line 45 updates the proposal only to a value V (different from ⊥) for which a COMMIT message with a valid evidence exists (guard in line 43). This implies a strong quorum of PREPARE messages for value V in a round, implying that some honest node (actually more than 1/3 of QAP) sent PREPARE for V.

Assume by contradiction that an honest node sent PREPARE for value V which breaks the invariant if some honest node updates its proposal to V and let r be the first such round.

We distinguish two cases:

1) r = 0 - the contradiction stems from the fact that a strong quorum of PREPARE msgs in round 0 for V implies that the invariant was already broken at some honest participant which is false due to Lemma Consistent Initial Proposals.
2) r > 0 - in this case, we show that a Byzantine participant could not have sent a CONVERGE message with valid evidence for value V. Note that this evidence requires a strong quorum of PREPARE messages for V in round r-1 including at least one from an honest node. Contradiction with the assumption that r is the first round in which an honest node sends PREPARE for V.

QED (Consistent Proposals)

Corollary. (Minimal Honest Proposal (MHP)) In each round, there is an honest participant with a proposal MHP that is a prefix of every other honest participant's proposal. If this holds for honest participant p in round r, we say p holds MHP in r.

Lemma. (Convergence to MHP) *If at the beginning of round r greater than a synchronous round r', p holds the MHP, and p obtains the lowest ticket of all participants in the CONVERGE step of r, and the system is Δ-synchronous, then all honest participants set their proposal to MHP before sending the PREPARE message m in r with m.value=MHP.*

Proof. (Convergence to MHP) By the Synchronous Round lemma, there is a synchronous round r, after GST, such that all honest participants reach r and no honest participant completes CONVERGE step until all honest enter r, and they spend together at least Δ in the CONVERGE step of this round. From this round onwards, honest participants operate in round lock-step (i.e., observe all of each other's CONVERGE messages) unless some of them decides.

Hence all honest participants receive p's CONVERGE message and execute lines 22-24, assigning MHP to value in line 22.Since MHP is in C (by (by initialization of C in line 12, QUALITY step lines 12-14 and Lemma Consistent Proposals) , all honest participants set their proposal to MHP in line 26. Hence all honest nodes send a PREPARE message m in round r with m.value=MHP.

QED (Convergence to MHP)

Lemma. (Same proposal + same PREPARE ⇒ decision) *If all honest participants have the same proposal equal to v and send <PREPARE, r, v> in the same round r > 0 and the system is Δ-synchronous, then all honest participants decide v in round r.*

Proof. (Same proposal + same PREPARE ⇒ decision) By Δ-synchrony, all honest participants receive all other honest participants' PREPARE messages in the PREPARE step of round r. Since honest participants form a >⅔ QAP quorum, the condition in line 32 is satisfied and all honest participants send a COMMIT message for v in round r. By Δ-synchrony all honest participants receive COMMIT messages from honest participants with >⅔ QAP and decide v.

QED (Same proposal + same PREPARE ⇒ decision))

Theorem. (Termination) *Every call to GossiPBFT implementation of F3 eventually returns with probability 1.*

Proof. (Termination)

By contradiction: Assume a non-Byzantine participant p never returns (with probability > 0).

We distinguish 2 Cases:

1. No non-Byzantine participant ever returns.
2. Some non-Byzantine participant p' other than p eventually returns.

Case 1: No non-Byzantine participant ever returns.

By the Advancing Rounds lemma, every non-Byzantine participant keeps advancing to higher rounds indefinitely.

Let r be the first round after GST such that:

- All non-Byzantine participants reached r after GST (see also this section).
- The non-Byzantine participant p that has the Minimal Honest Proposal (MHP) obtains the smallest tickets in the CONVERGE step of r. This event has non-negligible probability in every round (at least Power(p) for every round in which an honest participant p holds the MHP) and hence will eventually occur with probability 1 (with probability 0, such a round never occurs).

By Lemma Convergence to MHP all honest participants set their proposal to MHP and send PREPARE for MHP in round r.

By Same proposal + same PREPARE ⇒ decision lemma, all honest participants decide. A contradiction.

Case 2: Some non-Byzantine participant p' other than p eventually returns.

By line 41 and line 55, if p’ decides, it broadcasts a DECIDE message with evidence for a strong quorum of identical COMMIT messages. By BEBroadcast, p eventually receives this DECIDE message and processes it in lines 53-56 and decides as well. → A contradiction.

QED(Termination)

Validity

Theorem. (Validity) *If an honest participant decides c, then c is a prefix of the input chain of some honest participant.*

Proof. It is straightforward to observe that c must be prefixed by baseChain; namely, any QUALITY message carrying a proposal which does not have baseChain will be invalid and hence all subsequent messages by message validation. It is also straightforward to see that if exactly baseChain is decided, then Validity holds.

Assume now that chain c, heavier than baseChain, is decided by an honest participant, such that no honest participant proposed a chain with c as the prefix.

By ConsistentProposals lemma, no honest participant sends PREPARE for c in round=0, and hence c cannot be decided by an honest participant in round 0, as a strong quorum comprising more than 2/3rd of QAP for c cannot be established.

This implies that the proposal selected by an honest participant in line 45 of round=0 cannot be c. Moreover, in case a participant receives all COMMIT messages with value ⊥ in round=0, then a participant will retain its proposal set in the QUALITY step. If a Byzantine participant sends a CONVERGE message in round 1 with value c, and has the lowest ticket, no honest participant will have this value c in C and will not send a PREPARE for it. Therefore, c cannot be decided in round=1 either. This argumentation extends to future rounds by induction.

QED(Validity)

Agreement

Theorem. (Agreement) *No two honest participants decide different chains.*

Proof. We show that if an honest participant p decides a chain c in round r, no other honest participant p’ can decide a different chain c’ in round r’ ≥ r. Let us assume p is the first honest participant to decide. Consider, by contradiction:

Case 1: p’ decides c’ in the same round r’=r:

Since p decided c, then, by line 38, p received a strong quorum Qc of COMMIT messages for c in round r (*Power(M)* > ⅔, i.e., “67% of QAP”).

(1) Since we assume a strong quorum of more than 2/3rd of QAP of non-Byzantine participants Qh, strong quorums Qh and Qc intersect, and Qc contains at least a weak quorum of non-Byzantine participants WQhc (*Power(M, baseChain)* > ⅓, i.e., “34% of QAP”) who all sent COMMIT messages in round r for c.

Similarly, since p’ decided c’, there is a strong quorum Qc’ of COMMIT messages for c’ in round r’.

Since Qc’ and WQhc intersect in at least one non-Byzantine participant, a non-Byzantine participant sent COMMIT messages in round r for both c and c’. A contradiction.

Case 2: p’ decides c’ in the next round r’ = r+1:

First we prove a Lemma which gives a useful invariant for carrying over decided values in a previous round as a proposal in subsequent rounds.

Lemma 1. (decision implies mightHaveBeenDecided = True). If some non-Byzantine participant decides c in round r, then every mightHaveBeenDecided(c, r)=True for all correct participants.

Proof. (Lemma 1). If some non-Byzantine decided c in round r, then all participants receive at least one COMMIT for c analogously to case 1 shown above. Equivalently, there is at least a weak quorum of COMMIT messages for c from correct participants. As a result, for every set M of COMMIT messages received in round r such that Power(M)>= ⅔ , and for M' ← { m | m ∈ M AND m.value = c}, it follows that Power(M) <= ⅔ + Power(M'). This is equivalent to Power(M') + (1-Power(M)) >= ⅓, resulting, by line 60, in mightHaveBeenDecided returning True.

Lemma 2. (Non-Byzantine carry-over decided proposal). *If some non-Byzantine participant decides c in round r, then every non-Byzantine participant p’ updates its proposal variable to c at the end of round r.*

Proof. (Lemma 2) By (1), line 38 (the COMMIT step waiting for a strong quorum of COMMITs in every round) and non-empty intersection of weak quorum and any strong quorum awaited in line 39, it follows that every honest participant, including p’, received a valid COMMIT message for c in round r.

This implies that the condition in line 43 for p’ is TRUE (i.e., ∃ m ∈ M: m.value ≠ c), thanks to Lemma 1. Now we prove that there is no other value in M which could be chosen as the next round proposal in line 45.

Assume by contradiction this is the case and that there is m' ∈ M, such that m'.value=c' different from c and from ⊥. By message validity condition in line 39, m' is a COMMIT message with valid evidence which implies that m'.evidence contains a strong quorum of PREPARE messages in round r for c'. However, since p' also received a strong quorum of PREPARE messages in r for c, this implies that at least one non-Byzantine participant sent PREPARE for both c and c' in r. A contradiction.

QED (Lemma 2).

Lemma 3. (Byzantine carry-over decided proposal). *If some non-Byzantine participant decided c in round r, then every CONVERGE message m with valid evidence in round r+1 has m.proposal=c.*

Proof. (Lemma 3) Assume by contradiction a participant sends a CONVERGE message in r+1 for value c' different from c (by Lemma 2 the sender node must be Byzantine). To deem this message valid, its evidence needs to be for either:

- A strong quorum of COMMIT messages for ⊥ in r, implying at least one non-Byzantine participant sent COMMIT messages in r for both ⊥ and c (a contradiction), or
- A strong quorum of PREPARE messages in r for c', implying at least one non-Byzantine participant sent PREPARE messages in r for both c and c' (a contradiction).

QED (Lemma 3).

(continuing proof of Case 2)
Since p' decides c' in round r+1 (and not by receiving a DECIDE message), at least a weak quorum of non-Byzantine participants participates in round r+1. By Lemmas 2 and 3, all valid CONVERGE messages in round r+1 are for c. Hence all PREPARE messages by non-Byzantine participants in round r+1 are for c or ⊥. Consequently, no other value than c can be decided in r+1.

Case 3: p' decides c' in future rounds r' > r+1:

By induction: if p decided c in round r, then all valid CONVERGE messages in round r+1 are for c (Lemma 3). The same occurs with PREPARE messages in round r+1 (since the same evidence is used for PREPARE as for CONVERGE, for all rounds>0). As a result, the only valid COMMIT messages in round r+1 are for ⊥ or for c, with all correct participants sending COMMIT for c (since they received a strong quorum of PREPAREs for c from this round). This means that in all strong quorums of COMMITs for round r+1, there is at least one COMMIT for c, and that the statement of Lemma 1 is satisfied. As a result, the only valid CONVERGE in round r+2 is a CONVERGE containing an evidence with a strong quorum of PREPAREs for c, which we know all correct participants hold, from round r+1, because no strong quorum of COMMITs for ⊥ exists in round r+1. The proof follows then by induction for subsequent rounds.

QED (Agreement)

Proof of Finality

Theorem. (Proof of Finality) *If an honest participant finalizes (i,h,PoF), then PoF is a signature of 2/3rd QAP majority corresponding to PT(baseChain) input in instance i of some honest participant..*

Proof. By the algorithm, the PoF is the set of signed COMMIT messages whose senders form a strong quorum. By the definition of a strong quorum, the senders form a >⅔ QAP majority of the power defined by the power table of baseChain. By assumption, all non-Byzantine participants refer to the same baseChain.

QED(PoF)

Progress

Note that progress is shown under the assumption that all honest participants start an instance at most Δ apart. We note that this occurs after GST due to the way decision occurs, and the fact that starting a new instance is solely determined by a previous instance terminating: by line 41, the first honest to reach the DECIDE phase will send a DECIDE message. All honest participants will receive that DECIDE message and send their own DECIDE, by line 55. Also note that, since honest participants relay delivered messages in GossipSub, there is no distinction between valid messages from Byzantine participants and from honest participants once an honest participant delivers it. Thus, since these messages are all sent after GST and termination of the DECIDE phase requires a strong quorum, all honest participants terminate instance *i* at most Δ apart, and therefore start the next instance preserving this distance.

Theorem (Progress)

If the system is Δ-synchronous, i.e.:

- All honest participants can communicate within a known time bound Δ, and
- All honest participants invoke F3(i, *, *) at most Δ apart from each other,

Let c be the heaviest common prefix of the inputs of all honest participants in instance i. Then, if an honest participant finalizes (*i'*,*c'*,*), then *c is a prefix of c'* with probability > ⅔.

Proof. (Progress) By Δ-synchrony, a non-Byzantine participant p will receive all other non-Byzantine participants' QUALITY messages in the QUALITY step (>⅔ QAP). Let M be the set of QUALITY messages received by p in the QUALITY step.

By definition of the StrongQuorum predicate and the fact that the input of all non-Byzantine participants is prefixed with c, for no chain heavier than c that is not prefixed by c can StrongQuorum(c,M) hold (i.e., such a chain cannot be backed by >66% QAP). By definition of StrongQuorum(), p sets its proposal to a chain prefixed by c in line 13. Therefore

(2) Every non-Byzantine participant sets its proposal variable to an extension of c by the end of the QUALITY step.

There are 2 cases for when p can finalize c’.

1. p finalizes c’ in round 0.
2. p finalizes c’ in a round r > 0.

Case 1: p finalizes c’ in round 0.

If p finalizes c’ in round 0, then p must have received a strong quorum of COMMIT messages for c’ in round 0. Therefore, by (1), at least a weak quorum of honest participants sent a COMMIT message for c’ in round 0.

Similarly, by line 32 and (1), at least a weak quorum of honest participants sent a PREPARE message for c’ in round 0.

Since, in round 0, the value of each non-Byzantine participant’s PREPARE message is set in the QUALITY step (line 14), c’ is an extension of c.

Case 2: p finalizes c’ in a round r > 0.

Consider the case in which a non-Byzantine participant p1 obtains the lowest ticket in rounds r=1. This occurs with probability at least 2/3.
In round 1, it is easy to argue that p1 sends a CONVERGE message m with m.value such that isPrefix(c,m.value). It is also easy to see that m.value needs to be a proposal of some honest node in round 0.

Since the system is Δ-synchronous, in EC m.value has reached every honest participant, which invoked ECupdate(chain) and added m.value to the ECcompatiblechains variable.

Therefore, for every honest node, condition in line 25 holds and every honest node changes its proposal to m.value and every honest node sends PREPARE with m.value. By Lemma (Same proposal + same PREPARE ⇒ decision) every honest node decides m.value by end of round 1.

QED(Progress)

EC Compatibility

Theorem. (EC Compatibility) *An honest node never contributes to forming a 2/3rd QAP (super-majority) of votes for a chain c which its local EC instance did not locally already deliver, unless it already observes evidence of such a super-majority for chain c.*

Proof. (EC Incentive Compatibility) Follows directly from QUALITY step lines 12-14 (votes in PREPARE in round 0 must be a prefix of own inputChain, which is delivered by local EC), from lines 23-24 by which a sway of a local proposal follows a valid strong quorum of PREPARE messages, lines 53-56 (sway in line 26 must be supported by EC locally delivering that value earlier) , and lines 26, 45 by which a sway of a local proposal follows a valid CONVERGE/COMMIT message which in turn requires evidence consisting of PREPARE message for a value from a super-majority (>⅔ QAP), and only if the sway is necessary as the value might have been decided by some other non-Byzantine participant (lines 23 and 43).

QED (EC Compatibility)

Appendix A: GossiPBFT w/o piggybacked evidence (assumes Reliable Broadcast)

Protocol changes

Our Network Model models GossipSub as best-effort broadcast with the following properties:

- If p and p’ are honest, then every message broadcast by p is eventually delivered by p’.
- No message is delivered unless it was broadcast by its sender.

This holds in GossipSub, because even if there might be peers in-between p and p' that are attempting to hold back messages (eclipse), GossipSub will isolate them and effectively disconnect them from the network.

Because of its implementation, GossipSub could arguably be modeled and reasonably be assumed (cf. discussions with ProbeLab and empirical experience) to satisfy properties of reliable broadcast (RBroadcast) with the following *additional* property:

- For any message m, if an honest participant delivers m, then every honest participant delivers m.

With this assumption, we can streamline GossiPBFT by:

- replacing every BEBbroadcast invocation with RBroadcast,
- omitting the evidence field in all messages (and rely on the additional property of RBroadcast to provide such evidence) and
- conduct message validation based on locally received valid messages at a participant as follows (Valid() predicate stays the same):

Every honest participant maintains two sets:

- R: The set of all received messages
- V: The set of all validated messages

Only messages in V are considered by the main algorithm. To update the sets R and V, every honest participant executes the following algorithm concurrently with the main algorithm.

Upon reception of a message m:

- $R \leftarrow R \cup \{m\}$
- while $\exists$ $m' \in R \setminus V$: Valid(m’)
    - $V \leftarrow V \cup \{m'\}$

Note that a participant only needs to retain the sets R and V for the lifetime of the corresponding GossiPBFT instance. After a value has been finalized and the finalization proof broadcast to other participants, all data related to the GossiPBFT instance, including the sets R and V, can be garbage-collected. **Also note that we do *not* filter out equivocations in the context of message validation**. Namely, if a Byzantine, equivocating participant sends multiple messages in the same step in the same round with different values / proposals, we must add them both to V (provided they fulfill the above criteria). Any of them might be relied on by an honest participant and thus must also be available to all other honest participants.

ValidEvidence(m):

```
(m = < step, value, round, ticket>                                | valid CONVERGE or
AND (step = CONVERGE OR (step = PREPARE and round>0)    | PREPARE for round>0
needs
AND (∃ M⊆V: Power(M)>⅔}                                           | a strong quorum of
valid
        AND ((∀ m' ∈ M: m'.step = COMMIT AND m'.value = ⊥)  | COMMIT msgs for ⊥
                OR (∀ m' ∈ M: m'.step = PREPARE AND              | or PREPARE msgs for
                        m'.value = m.value))                              | CONVERGE value
        AND (∀ m' ∈ M: m'.round = m.round-1)))                   | from previous round

OR (m = <COMMIT, value, round>                                   |valid COMMIT needs
    AND ((∃ M⊆V: Power(M)>⅔                                     |a strong quorum of valid
                AND ∀ m' ∈ M: m'.step = PREPARE                           | PREPARE
        messages
                AND ∀ m' ∈ M: m'.round = m.round                          | from the same round
                AND ∀ m' ∈ M: m'.value = m.value)                         | for the same value,
        or
        OR (m.value = ⊥))                                                | COMMIT is for ⊥ with
                                                                          | no evidence
```

The DECIDE step can be entirely omitted (hence its case is omitted above from ValidEvidence()) with enabling COMMIT on a strong quorum of messages with the same round and value at any time, while also omitting RBbroadcast for any DECIDE. The full cleaned-up version of this protocol variant is given below.

F3(inputChain, baseChain) returns (chain, PoF):

```
1:      round ← 0;
2:      decided ← FALSE;
3:      proposal ← inputChain;  \* holds what participant locally believes should be a decision
4:      timeout ← Δ
6:      value ← proposal;        \* used to communicate voted value to others (proposal or ⊥)

8: while (not(decided))  {
10:     If (round = 0)
```

```
11:             BEBroadcast <QUALITY, value>; trigger (timeout)
12:             collect a clean set M of valid QUALITY messages
                until StrongQuorum(proposal,M) OR timeout expires
13:             Let C={prefix: isPrefix(prefix,proposal) AND StrongQuorum(prefix,M)}
14:             If (C = ∅)
15′:                    C ← {baseChain}
15:                     proposal ← baseChain \* no proposals of high-enough quality
16:             Else
17:                     proposal ← heaviest prefix ∈ C \* this becomes baseChain or sth heavier
18:             value ← proposal

20:     If (round > 0):                                                 \* CONVERGE
21:             ticket ← VRF(Randomness(baseChain) || round)
22:             value ← proposal        \* set local proposal as value in CONVERGE message
23:             BEBroadcast <CONVERGE, value, round, ticket>; trigger(timeout)
24:             collect a clean set M of valid CONVERGE msgs from this round
                until timeout expires
24′:            prepareReadyToSend ← False
25′:            while (!prepareReadyToSend){
25:                     value ← GreatestTicketProposal(M)    \* leader election
25′:                    if (value justified by a strong quorum of PREPAREs AND
mightHaveBeenDecided(value, r-1)):
26′:                            C ← C ∪ {value}
26:                     If (value ∈ C)
27:                             proposal ←value
28′:                            prepareReadyToSend ← True    \* Exit loop
28:                     Else
29:                             M = {m ∈ M | m.value != value}

30:     BEBroadcast <PREPARE, value, round>; trigger(timeout)
31:     collect a clean set M of valid PREPARE messages from this round
        until power(M)>2/3 AND (StrongQuorumValue(M) = proposal OR timeout expires)
32:     If (HasStrongQuorumValue AND StrongQuorumValue(M) = proposal)
33:             value ← proposal                \* vote for deciding proposal (COMMIT)
35:     Else
36:             value ← ⊥               \* vote for not deciding in this round

40:     BEBroadcast <COMMIT, value, round>; trigger(timeout)
41:     collect clean set M of valid COMMIT messages from this round
        until (HasStrongQuorumValue(M) AND StrongQuorumValue(M) ≠ ⊥)
                OR (timeout expires AND Power(M)>⅔)
44:     If (HasStrongQuorumValue(M) AND StrongQuorumValue(M) ≠ ⊥)
45:             M ← {m ∈ M : m.value = StrongQuorumValue(M)} \* filter M
47:             decided ←TRUE
48:             return (StrongQuorumValue(M), M)
49:     If ( ∃ m ∈ M: m.value ≠ ⊥ s.t. mightHaveBeenDecided(m.value, r)) \* sway proposal to a
potentially decided value
50':            C ← C ∪ {m.value}               \* add to candidate values if not there
50:             proposal ← m.value;
54:     round ← round + 1;
55:     increase(timeout)}  \*end while
```

Concurrently with the above, every participant enables, at any point in time, a decision based on a strong quorum of COMMIT messages pertaining to the same value from the same round, regardless of the round number.

```
                                                            \* decide anytime
60:     upon reception of a clean set of valid <COMMIT, v, r,> messages M:
61:          (∀ m∈M: (m.value=v AND m.round=r)) AND  Power(M)>⅔ AND not(decided)
62:             decided ← TRUE
63:             return (v, M)
```

Comparison to the evidence-based version

Compared to the evidence-based protocol, this version avoids evidence piggybacking to protocol messages while assuming GossipSub to provide reliable broadcast properties. The choice between two versions for actual implementation is driven by tradeoffs summarized in the following table. Below, n is the number of consensus participants.

GossiPBFT version	*GossipSub assumption*	*Code pros*	*Code cons*	*Message sizes*
w/ evidences	BEBroadcast (weaker)	Simpler, embedded message validation	Involves signature aggregation; broadcast DECIDEs	O(n) bits (worst case)
w/o evidences	RBroadcast (stronger)	omits signature aggregation; omits broadcasting DECIDEs; message format slightly	more complex msg validation code (recursive)	O(1)

	🟨	🟩 simpler	🟨	🟩

Appendix B: Beyond 33% QAP adversary - Filecoin power leakage (FPL) attack

In GossipPBFT, a strong adversary, which controls more than 33% QAP can violate properties of F3, including halting F3/GossipBFT (violating Progress and Termination) and even making several tipsets finalized, violating Agreement. The latter attack on Agreement is more difficult if the adversary controls between 1/3 and 1/3 QAP, when adversary needs to also control the network partition, and easier in case of >2/3 QAP adversary in which the adversary could practically fully control F3/GossiPBFT.

Also, controlling beyond ⅓ QAP, the adversary could mount a combined attack on F3 and EC, called Filecoin power-leakage (FPL) attack. However, unlike for off-the-shelf BFT consensus protocols, where merely controlling more than 1/3 QAP suffices for an adversary to mount the FPL attack (see details here), with GossiPBFT this is more difficult and involves the following:

1) strong adversary controlling more than 1/3 QAP.
2) honest participant input chain partitioning.

Below we illustrate the specific variant of the FPL attack which applies to GossiPBFT, followed by a mitigation discussion.

FPL attack setup on GossiPBFT

Recall that the duration of a window to post a specific subset of WindowPosts in Filecoin/EC is 60 epochs. For simplicity, we assume this BFT instance is the last one of a window (all chains proposed as input by honest participants have head tipsets of the last epoch of a window). Let us define the first EC epoch of the new window *e_start* and the last EC epoch *e_end = e_start+59,* i.e. all honest proposals (input chains) have head tipsets from epoch e_start-1. The goal of the adversary is to manage to finalize its private fork for all 60 epochs of the window *[e_start, e_end]*. For this, the adversary proceeds as follows:

Consider the following execution of GossiPBFT:

1) Setup
    - **Honest participants are partitioned by input value into two sets,** P1 and P2, that differ on the input chain, **each holding at least X% of the QAP**: W.l.o.g., we assume the input chains of P1 and P2 are as follows ([] denotes an extension to baseChain, letters denoting tipsets):
        - P1: chain C = [a b c] (A new suffix chain of baseChain with three tipsets a b and c)
        - P2: chain D = [a b d] (A new suffix chain of baseChain with three tipsets a b and d)

- **Adversary controlling fraction Y of QAP, where ⅔ >Y>⅓ , such that X+Y>⅔** (adversary with Y>⅔ can control F3 at will)**.** Controlling more than ⅓ QAP allows the adversary to temporarily halt GossiPBFT.
- As an illustration, with Y≲ ½ (roughly 50% adversary), X (size of honest participant partitions) needs to be larger than ⅙ QAP (X≳⅙)

2) QUALITY step: The adversary **speculatively selects** one of the two input values, assume w.l.o.g. D, and sends Q2=<QUALITY, D>. Participants from P2 and adversary end QUALITY step with proposal D, whereas those from P1 end with baseChain or D.
3) The adversary and honest participants from P2 send also the corresponding PREPARE for proposal D get the strong quorum and proceed to begin the COMMIT phase.
4) Here, the adversary stops participating temporarily in the GossiPBFT protocol. As Y>⅓, GossiPBFT cannot terminate.
5) Adversary turns to EC and waits to see which of the two forks, whether C = [a b c] or D = [a b d], becomes the heaviest one. If D=[a b d] becomes the heaviest one, the adversary cannot continue performing the attack and "allows" D to be finalized by GossiPBFT. If instead C = [a b c] becomes the heaviest one in EC, the attack continues. After enough time, this fork has multiple more tipsets from newer EC epochs, say for example [a b c e f ... z]. Let e (resp. z) be the tipset built at epoch e_start (resp. e_end) by honest participants.
6) Then, the adversary privately builds a fork that extends [a b d], for example : [a b d e' f' ... z']. (They can do this locally as these epochs have already passed). Let e' (resp. z') be the tipset built at epoch e_start (resp. e_end) by selfish and Byzantine participants. The goal of the attackers will be to make this private fork the heaviest one.
7) Adversary then sends COMMIT messages for D = [a b d] in GossiPBFT. D gets finalized with COMMIT messages for D from P2.
8) As D [a b d] has been finalized, the updated fork choice rule of EC at each honest participant chooses [a b d] as the heaviest chain. This means that all the work done by honest participants in epochs [e_start, e_end] and that led to chain [a b c e f ... z] has become deprecated in favor of the chain D=[a b d] that has been finalized by GossiPBFT.
9) Now, attackers release their private fork [a b d e' f' ... z']. Honest participants cannot build a heavier fork by following the protocol, since those epochs have already passed. All honest participants recognize [a b d e' f' ... z'] as the heaviest chain.
10) In BFT instance i+1, all participants propose the input chain [a b d e' f' … z'], that gets finalized. Consequently, the adversary has successfully censored correct participants during all 60 epochs in a window, performing a power leakage.

As we see this attack is considerably more elaborate than FPL on off-the-shelf BFT.

Mitigation strategies

With respect to strong adversary FPL attacks on GossiPBFT, we propose one or more of the following

1. **"Pulling the handbrake": locally stop participating in GossiPBFT instances once one instance takes too long and EC progresses for "too long".**

   The handbrake is a locally defined amount of time (let T be the system parameter defining the amount of time an honest participant considers 'too long') after which an honest participant will stop participating in instances of GossiPBFT. The participant will still adopt decided values if other participants keep participating in instances. Once enough participants stop participating, they can resort to social consensus to reorganize and update Filecoin client software to obtain fast finality again.

   This solution does not entirely solve the problem, as the attackers can divide the attack into multiple smaller attacks, each requiring less time than T. The only requirement the attackers have to perform the attack is that in each sub-attack at least three epochs take place during the time in which they halt termination. This solution is however an effective mitigation strategy.

   A critical parameter for this solution is the concrete value T to wait for in order to pull the handbrake. On the one hand, we want a sufficiently long T to ensure not pulling the handbrake during real network congestion. On the other hand, we want to make sure T is small enough to prevent this attack. The concrete value is not straightforward to calculate and might require an analysis of the likelihood of attackers mounting multiple, small-scaled, consecutive attacks, and the amount of epochs that must be ensured to hold some honest proposals in a window to guarantee all WindowPoSts are included.

2. **Offset head tipset to a lookback parameter**. Offset the head tipset of the input chain proposed in GossiPBFT. That is, honest participants look at the current EC epoch, and propose the head tipset T epochs back. This increases the probability that there will not be a partition of honest between P1 and P2, and eliminates the possibility of an attacker selectively releasing a block to create the partition at will. Also, if combined with a synchrony assumption, and making T big enough, then full censorship resistance is provable. The drawback is that, if applied, this mitigation delays Filecoin finality.