F3: GossiPBFT message rebroadcast (v3)

Alejandro Ranchal-Pedrosa

Background: This document extends the latest specification by removing the need to ensure guaranteed delivery of all sent messages. There are two reasons that led to removing this assumption:

1- According to the latest specification (prior to this doc) of GossiPBFT. If some non-Byzantine participants lag behind in past instances/rounds (wrt to some other non-Byzantine participants), they need to buffer messages for the future instances/rounds. This introduces a simple DOS attack, specially for messages from future instances (since the stragglers cannot verify messages from future instances), by which an adversary causes a memory overflow of messages buffered (if queue of buffered messages is unbounded), or causes some messages from correct participants to be dropped (if queue is bounded).

2- Another reason why we need to deal with messages that may have been dropped is because GossipSub (the network layer we use) does not guarantee delivery of sent messages.

We scope the problem here and outline a specific solution for GossiPBFT, to then prove it correct.

Re-scoping the problem: Reformulating assumptions

In the FIP and related docs, we prove GossiPBFT correct in a partially synchronous model, where all messages are delayed indefinitely but ultimately delivered by (i.e. received by) all correct participants. The problem stems from this assumption, which requires messages to be stored indefinitely as soon as received, especially those that cannot be validated (because they are from future instances). This means that a solution cannot rely (solely, at least) on using a queue of messages at the receiver (because it would need to be an unbounded queue). A bounded queue, however, opens the path to a DOS attack by sending many messages, causing messages being dropped at the receiver before delivering them. In addition to that, the network layer used by F3, GossipSub, does not provide guaranteed delivery of sent messages.

As partial synchrony seems, thus, too abstract to represent this problem, we restate the network assumption. Instead of partial synchrony, in which all messages are eventually delivered, we assume that an arbitrary number of sent messages can be dropped by the network (not being delivered) initially, up until a Global Stabilization Time (GST) time when messages are guaranteed to be delivered (i) by all participants and (ii) within the synchronous timeout. The latter (ii) is one of the two formulations of partial synchrony, while the former (i) is a novel restriction that makes the assumption weaker than partial synchrony. Let us refer to this network model as "partial synchrony with lossy channels".

The problem is thus to solve repeated consensus for all correct participants in this network model. If we have a protocol that tolerates the temporary loss of messages until some unknown GST time, then we have a protocol that can tolerate an arbitrary number of messages being dropped by being received too early wrt to the local state, and that preserves correctness when using GossipSub as the network layer.

Status Quo: Pseudocode

For completeness, we start by specifying the pseudocode of a single instance as stated in the latest specification prior to this doc:

GossiPBFT(instance, inputChain, baseChain, participants) returns (chain, PoF):

```
1:  round ← 0;
2: decideSent ← False;
3: proposal ← inputChain;  \* holds what participant locally believes should be a decision
4: timeout ← 2*Δ
5: value ← proposal     \* used to communicate voted value to others (proposal or ⊥)
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 ∪ {prefix: isPrefix(prefix,proposal) AND StrongQuorum(prefix,M)}
13:              proposal ← heaviest prefix ∈ 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
mayHaveStrongQuorum(value, r-1)):
```

24: C ← C ∪ {value}
25: If (value ∈ C)
26: proposal ←value * we sway proposal if the value is incentive 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. mayHaveStrongQuorum(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.
                                                                \*Helper function
57: mayHaveStrongQuorum(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:     if Power(M') + (1-Power(M)) < ⅓ : \* value cannot have been decided
61:             return False
62:     return True
```

And the corresponding f3 pseudocode:

```
63: // finalizedTipsets is a list of records with tipset and PoF fields representing
64: // all finalized tipsets with their respective proof of finality.
64: // It is initialized with the genesis tipset by definition, which needs no proof.
65: finalizedTipsets[0].tipset ← genesis
66: finalizedTipsets[0].PoF ← nil
67: i ← 1
68: while True:
69:     participants ← PowerTable(finalizedTipsets[i-1].tipset)
70:     proposal ← ChainHead()
71:     finalizedTipset, PoF ← GossiPBFT(i, proposal, finalizedTipsets[i-1], participants)
72:     finalizedTipsets[i].tipset ← finalizedTipset
73:     finalizedTipsets[i].PoF ← PoF
74:     i ← i + 1
```

Before we dive into the proposed solution, we give the intuition to the proposed solution by incrementally addressing the issues that arise with the aforementioned pseudocode when considering lossy channels and DOS-attacks:

1st issue: DOS-attack causing an overflow of the queue of buffered messages

As mentioned at the beginning of this document: participants can receive messages from a future instance (for which they do not have the power table, as per line 69 the power table for instance i is calculated after deciding at instance i-1). According to the above-shown code, since messages are only broadcast once, these messages need to be buffered until the participant can deliver (i.e. process) them. This opens a DOS attack vector: attackers send many messages for future instances, causing the queue of buffered messages to grow to the point of a memory overflow. Even within instances, attackers can send many COMMIT messages for $\perp$ (which carry no evidences) for future rounds, having the same effect. If messages from correct participants end up being dropped, a correct participant could be stuck waiting for a strong quorum of messages in lines 31, 39 or 51.

A potential solution for this attack alone, if the network layer guaranteed delivery of all sent messages, involves the receiver selectively dropping messages when the buffer is full. This can guarantee correctness of just one instance while keeping the buffer with a bounded size that remains practical, when paired with a strategy for the receiver to skip rounds when receiving enough messages for a future round (more details in our proposed solution**).**

While this solution solves the problem within an instance, the attacker can still generate messages for future instances that cannot yet be validated, and need to be buffered.

Note: still, a solution to deal with messages from future instances can be introducing a synchrony assumption that limits how many instances in the future the fastest correct participant can be relative to the slowest correct participant, paired with an offset in the application of the power table by at least that many instances. That is, the state resulting from the output of instance *i* only comes in effect wrt to the power table in instance *i+offset*, where *offset* is precisely the assumed max distance between the slowest and fastest correct participant (in number of instances). This way, a correct participant can discard messages if they are *offset* instances in the future with respect to their current instance, or if they are invalid according to the power table, or selectively drop them otherwise. We discard this solution in this document as it is insufficient because of the following 2nd issue.

We discuss how to extend the above solution to deal with loss of messages.

2nd issue: Network layer (GossipSub) does not guarantee delivery of all sent messages

If an arbitrary number of messages can be lost before GST, it is easy to see that, because in the above code messages are only broadcast once and never requested by the receiver, participants can get stuck waiting for a strong quorum in either PREPARE, COMMIT or DECIDE phases. The solution for this is either a push mechanism to rebroadcast messages or a pull

mechanism to request messages. We propose in this document a push mechanism to rebroadcast messages if participants are stuck within its current instance, but a pull mechanism for the finality certificates of instances that terminated with a decision already. We specify in this document the push mechanism for messages within a running instance, but omit the pull mechanism for finality certificates, as that is the scope of the finality exchange protocol (and not the scope of this document). We now propose the solution.

Solution: GossiPBFT with lossy channels

As seen above, our proposed solution has two elements:

1. A periodic rebroadcast of sent messages within an instance
2. A mechanism to selectively drop received messages within an instance and skip directly to a future round

Both are orthogonal in that they solve intertwined, but different issues: periodic rebroadcast is necessary to deal with lossy channels (issue 2), and selectively dropping messages and skipping to future rounds is necessary to tolerate DOS-attacks and not need unbounded buffers before GST (issue 1). This clarification is made in the effort to properly scope the solution in case our assumptions change (i.e. in case the network layer assumes reliable broadcast, or DOS-attacks, stop becoming a concern).

Pseudocode: GossiPBFT with lossy channels

We illustrate here below the proposed changes to the pseudocode as suggestions. We explain them here:

1. A **new timeout *timeout_rebroadcast***, that is at least slightly greater than the timeout of a step, after which, if the current step cannot be terminated, there is a rebroadcast of all the messages in the current round. This timeout is initialized in line 5', updated in line 50' and (iteratively if stuck) line 83, and triggered in lines 66 and (iteratively if stuck) 84.
2. A **wrapper of the function BEBroadcast into a new function reBroadcast**, that calls BEBroadcast, but also buffers sent messages (line 64) in a new buffer broadcast_msgs (line 8'). These sent messages are buffered so that a participant rebroadcasts all the messages of its current round if he is stuck, along with the COMMIT and PREPARE of the previous round. This function (reBroadcast) is replaced by BEBroadcast in lines 30, 38, 41 and 55. The QUALITY and CONVERGE BEBroadcast (line 10, 18) do not need to be replaced: nobody can get stuck in the QUALITY or CONVERGE phase. However, participants do rebroadcast their CONVERGE for the round if stuck in PREPARE/COMMIT (line 80), so that, in the case somebody needs to jump forward to the round, they can execute that round in full (from CONVERGE, updating their candidate set C if needed, line 106). In fact, participants rebroadcast all messages of their current round if stuck, except for QUALITY in round 0 (lines 70-82).

3. A **queue of received messages**, ***receiver_queue**,* which stores messages received for this instance (line 6'). Messages are added to the queue if they are for a future round than the current round (line 99). Note the queue considers buffering QUALITY messages, but they only need to be buffered if the participant is not already running the instance (as QUALITY is the first instance), which may seem out of the scope of this doc (as we mentioned we consider only one instance). It is however important to buffer QUALITY messages for the next instance, so that something else than baseChain can be decided after GST. So that QUALITY messages can be validated, we propose offsetting the power table state by one instance at least, as previously discussed, and buffering QUALITY messages for the immediately next instance. Messages in the receiver_queue are then delivered when reaching the corresponding step and round (lines 88 and 90). Although this queue can have unbounded size, thanks to the trim function dropping spammable messages (COMMITs for ⊥), we expect the sizes to remain manageable across participants. We assume in the proofs that the queue is bounded in size, and note that the queue can be made bounded by only keeping all messages for the current and two greatest known rounds at the expense of a more elaborate trim function (see v2 of this document).
4. A procedure for the receiver **queue to selectively drop messages** in the call to Trim() (line 100). How Trim() selects which messages to keep is explained in natural language at the end of the pseudocode. Upon reception of a message, the participant sees if it can deliver the message for this instance (either DECIDE or for this round and step), and delivers the message if that's the case (lines 92-97). Otherwise, the message is added to the queue (line 99).
5. A **procedure to jump forward to a future round (skipping intermediate rounds).** The triggering factor to jump forward to a future round is embodied in the function *shouldJump,* which is checked in line 101. shouldJump returns a CONVERGE msg for the future round to which to jump if the participant must jump forward to a future round (nil otherwise), which occurs only after gathering at least one valid CONVERGE and a weak quorum of valid PREPARE messages for a future round (the same round) (lines 114). Jumping forward never occurs in QUALITY nor in DECIDE (these phases are executed in full, line 111).
6. A **procedure to skip the timeout of a phase** if the PREPARE or COMMIT can be executed deterministically already (by the already received messages in that phase), that helps stragglers catchup to the phase executed by faster participants (modifications at lines 31, 39)
7. The rest of the changes outlined are cosmetic.

GossiPBFT(instance, inputChain, baseChain, participants) returns (chain, PoF):

1: round ← 0;
2: decideSent ← FALSE;
3: proposal ← inputChain; * holds what participant locally believes should be a decision
4: timeout ← 2*Δ
5': timeout_rebroadcast ← timeout + 1 * at least >timeout, how much greater is up to the participant to decide locally

```
5: value ← proposal     \* used to communicate voted value to others (proposal or ⊥)
6': receiver_queue ←newQueue(size: 7*n) \* bounded queue, where n is the number of
participants.
6: evidence ← nil       \* used to communicate optional evidence for voted value
7: C ← {baseChain}
7': step ← nil
8': broadcast_msgs ← {} \* map with latest msgs sent by this participant per step

8: while ( step != DECIDE )  {
9:      If (round = 0)
10:             BEBroadcast(<QUALITY, value>); trigger (timeout)
11':            updateStep(round, QUALITY)
11:             collect a clean set M of valid QUALITY messages
                until StrongQuorum(proposal, M) OR timeout expires
12:             C ← {prefix: isPrefix(prefix,proposal) AND StrongQuorum(prefix,M)} ∪ C
13:             proposal ← heaviest prefix ∈ C \* this becomes baseChain or sth heavier
14:             value ← proposal

15:     If (round > 0):                                                 \* CONVERGE
16:             ticket ← VRF(Randomness(baseChain) || round)
17:             value ← proposal        \* set local proposal as value in CONVERGE message
18:             BEBroadcast(<CONVERGE, value, round, evidence, ticket>); trigger(timeout)
18':            updateStep(round, CONVERGE)
19:             collect clean set M of valid CONVERGE msgs from this round
                until timeout expires
20:             prepareReadyToSend ← False
21:             while (!prepareReadyToSend){
22:                     value, evidence ← GreatestTicketProposal(M)         \* leader election
23:                     if (evidence is a strong quorum of PREPAREs AND
mayHaveStrongQuorum(value, r-1, COMMIT, ⅓ )):
24:                             C ← C ∪ {value}
25:                     If (value ∈ C)  \* see also lines 70-73
26:                             proposal ←value
27:                             prepareReadyToSend ← True    \* Exit loop
28:                     Else
29:                             M = {m ∈ M | m.value != value AND m.evidence.value !=
evidence.value} }


30:     reBroadcast(<PREPARE, value, round, evidence>); 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 OR NOT mayHaveStrongQuorum(proposal, r, PREPARE, 0))

32: If (HasStrongQuorumValue AND StrongQuorumValue(M) = 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: reBroadcast(<COMMIT, value, evidence, round>); **trigger**(timeout)
39: **collect** a clean set M of valid COMMIT messages from this round
until (HasStrongQuorumValue(M) AND StrongQuorumValue(M) ≠ ⊥)
OR (timeout expires AND Power(M)>2/3) OR (NOT mayHaveStrongQuorum(value, r, COMMIT, 0) for all value ≠ ⊥)
40: If (HasStrongQuorumValue(M) AND StrongQuorumValue(M) ≠ ⊥) * decide
41: reBroadcast <DECIDE, StrongQuorumValue(M), Aggregate(M))
43: Else if (∃ m ∈ M: m.value ≠ ⊥ s.t. mayHaveStrongQuorum(m.value, r, COMMIT, ⅓))
* m.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 inherited 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 * 2
50': timeout_rebroadcast ← max(timeout+1, timeout_rebroadcast) * update timeout_rebroadcast to be at least greater than timeout} *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, v, r, evidence> AND step != DECIDE
55: re-Broadcast (<DECIDE, v, r, evidence)
56: go to line 51.

*Helper function, if advPower = 0 then participant considers whether it is still possible for him to locally reach a strong quorum. If advPower = ⅓ , then participant considers whether any participant may reach a strong quorum in the presence of up to ⅓ equivocations.

```
57: mayHaveStrongQuorum(value, round, step, advPower):
58:     M ← { m | m.step = step AND m.round = round AND m is valid}
59:     M' ← { m | m ∈ M AND m.value = value }
60:     if Power(M') + (1-Power(M)) < ⅔ - advPower: \* value cannot have been decided
61:         return False
62:     return True

63: func reBroadcast(msg):
64:     broadcast_msgs[msg.round][msg.step] = msg
65:     BEBroadcast(msg)
66:     trigger(timeout_rebroadcast)
67:     updateStep(msg.round, msg.step)
68:     upon timeout_rebroadcast expires:
69:         If step = msg.step AND msg.round = round AND msg.instance = instance \* stuck, need to rebroadcast
70:             switch (msg.step) {
71:                 case DECIDE:
72:                     BEBroadcast(broadcast_msgs[0][DECIDE]) \* only rebroadcast DECIDE
73:                     break;  // Exit, no cascading
74:                 case COMMIT:
75:                     BEBroadcast(broadcast_msgs[msg.round][COMMIT]) \* no break, cascade to broadcast PREPARE, CONVERGE
76:                 case PREPARE:
77:                     BEBroadcast(broadcast_msgs[msg.round][PREPARE]) \* no break, cascade to broadcast CONVERGE
78:                 default: \* QUALITY or CONVERGE, which will never reach this point, but send more messages here because of cascading effect of switch cases
79:                     if msg.round > 0:
80:                         BEBroadcast(broadcast_msgs[msg.round][CONVERGE])
81:                         BEBroadcast(broadcast_msgs[msg.round-1][COMMIT]) \* broadcast COMMIT from previous round
82':                        BEBroadcast(broadcast_msgs[msg.round-1][PREPARE]) \* broadcast PREPARE from previous round
```

```
82:                     }
83:                     update(timeout_rebroadcast); \* increase and trigger again timeout_rebroadcast
84:                      trigger(timeout_rebroadcast)

85: func updateStep(round, new_step):
86:     step ← new_step
90:     receiveAll(receiver_queue.PopAll(round, step)) \* deliver all queued messages in the queue for this instance, round and step, and remove from queue


91: upon reception of a valid msg: \* assume validity implies same instance as currently running, different instances treated elsewhere (not scope of this doc)
92:     if msg.round < round \* drop message
93:         return
94:     else if msg.step = DECIDE
95:         deliver(DECIDE) \*DECIDE immediately delivered within an instance
96:     else if msg.step = step and msg.round = round
97:         deliver(msg) \* can be delivered as it is for this round and step
98:     else if msg.round > round
99:         receiver_queue.Add(msg)
100:        receiver_queue.Trim() \* selectively drop messages if bound met. See text below.
101:        if  (msg' ← shouldJump(round, step) s.t. msg' != nil)     \* one of the rest of conditions to be able to jump
102:                    round ← msg'.round;
103:                    timeout ← 2*Δ*2**msg'.round
104:                    timeout_rebroadcast ← max(timeout+1, timeout_rebroadcast)
105:                    if msg'.evidence.step = PREPARE: \* either this or strong quorum of COMMITs for ⊥
106:                        C ← C ∪ {msg'.value}          \* add to candidate values if not there
107:                        proposal ← msg'.value;              \* sway local proposal to possibly decided value
108:                    evidence ← msg'.evidence      \* strong PREPARE quorum is inherited evidence for the value (it exists because the value might have been decided)
109:                    jump to line 16  \* start new round by jumping forward

110: func shouldJump(round, step):
111:     if step = QUALITY OR step = DECIDE: \* never jump on DECIDE or QUALITY
112:         return nil
114:     if (∃ msg' ∈ receiver_queue st. msg'.step = CONVERGE          \* there must be a CONVERGE for the new round
            AND msg'.round > round                                    \* round must be greater
```

AND ∃ M ⊆ receiver_queue s.t. M is clean and valid and contains a weak quorum of PREPAREs for msg'.round)
116: return msg'
117: return nil

The function receiver_queue.Trim() contains a queue that keeps all messages in the current round, all validated QUALITY messages (as QUALITY from future instances may be valid but the participant may not have locally the power table that must be used to validate it), and all valid messages carrying a justification, but can drop all other messages (i.e. it drops COMMITs for ⊥ not in the current round).

QUALITY messages in the queue can be removed once the QUALITY phase is executed. Also, once a valid DECIDE message is received, all other messages in the queue for this instance can be removed.

Proof of correctness

We show correctness in the following proof, building upon the proofs that already showed correctness without arbitrary loss of messages. We omit mentions to the properties of Progress because progress is conditional to there being synchrony, and to Proof-of-Finality as this proof obviously remains unchanged: DECIDE steps are always executed for all participants and instances, thanks to rebroadcast, and regardless of whether they jump or not.

For (theoretical) completeness on these proofs we note that the property of termination should be considered the property of decision: participants need to always listen to received messages in order to share their finality certificate if somebody requests it. As mentioned above in this document, we do not specify the Finality Exchange Protocol (FEP) in this document. However, for completeness of the theoretical proofs, let us assume in this document the following naive approach for the FEP: a participant's SMR stores forever finality certificates, and periodically broadcasts them all. The receival of a valid set of finality certificates causes the participant to finalize all instances and start a new instance following the latest valid finality certificate received. This way, we can focus in the proofs for the case in which all participants are in the same instance.

Lemma 1' shows the good case is not affected by any of the modifications made in this document (same execution). Then we proceed to prove correctness with lossy channels, for which we first prove the following Lemma 2'.

Lemma 1'. (GST => no jump) If an instance starts after GST and there is Δ-synchrony (i.e. all participants start the instance at most Δ apart), then no correct participant jumps forward or drops any message from other correct participants.

Proof. By Δ-synchrony, all correct processes terminate all steps, starting with QUALITY, at most Δ apart, since a correct participant receives the messages from all correct participants at most Δ apart. Jumping occurs after gathering at least one CONVERGE and a weak quorum of PREPARE messages for a future round (the same round) (lines 114). We note that a weak quorum of PREPAREs implies at least one PREPARE from a correct participant. A PREPARE from a correct participant means that all other participants are at least already in the CONVERGE of the same round, by the fact that all participants are at most Δ apart and Δ-synchrony, meaning no jumping to a future round (and no messages from the current round are dropped).

Not dropping messages also follows from the fact that participants are at most one round apart thanks to Δ-synchrony, and the fact that the queue of messages keeps all justified messages and all messages in the current round.

QED

Lemma 2' (Once GST => one jump). Let r be the first round started by a correct participant once Δ-synchrony is met (and after GST) such that no other correct participant had started this round already, and suppose all participants are in a round other than the first round. Then, no correct participant jumps more than once (other than to the DECIDE phase).

Proof. We consider the worst case with the biggest distance (in phase and rounds being executed) between correct participants, and show that the fastest correct participants will send their PREPARE for a new round r' only once the slowest correct participants have terminated round r'-1. In the worst case, a participant jumps immediately when it receives a CONVERGE and a weak quorum of PREPAREs (lines 114). Let *timeout(r)* denote the timeout used for round r, that is, *timeout(r) = 2*Δ*2^r*. For a correct participant p to be at round r, it must have received a strong quorum of COMMITs for round r-1 (line 39), meaning at least a weak quorum of correct participants are at least at the COMMIT phase of round r-1. The rest of these correct participants may still be waiting for the timeout of phase COMMIT at round r-1, i.e. timeout(r-1). Eventually, however, this weak quorum can proceed to round r, where p will be waiting at the PREPARE phase (since this phase only terminates with a strong quorum of PREPAREs, line 31, and the adversary controls less than a weak quorum). At the moment a weak quorum of correct participants send their PREPARE for round r, the following occurs:

1- Upon receiving these messages, p terminates the PREPARE phase of round r and starts the COMMIT phase of round r (line 32-39) with the help of Byzantines.

2- Upon receiving these messages, all participants not already in r jump to the CONVERGE phase of round r (by lines 98-109).

To prove that no more jumping is possible, note that all messages are now being received (after GST) for round r, meaning that the only possibility to jump is to receive a weak quorum of PREPAREs and a CONVERGE for a future round. As such, we just need to prove that by the

time the fastest participant p sends its PREPARE all stragglers are already at the CONVERGE phase for the same round. Therefore, as long as 3*timeout(r) <= timeout(r) + timeout(r+1) then by the time the fastest participant sends its PREPARE, all stragglers have finished the previous round. Therefore, 3*timeout(r) <= timeout(r) + timeout(r+1) <-> 3*2*Δ*2^r <= 2*Δ*2^r + 2*Δ*2^(r+1) <-> 3*2^r <= 2^r + 2^(r+1), *which is always true for r>=0*.

QED

The assumption that all participants are not in the first round is not to have to deal with the timeout of the QUALITY phase (as no participant jumps before executing the QUALITY phase). The proof is analogous with instead one additional jump only for all correct participants that had to execute the QUALITY phase by the time GST is reached.

Theorem. (Correctness with lossy channels) In partial synchrony with lossy channels, GossiPBFT satisfies consensus.

Proof. All proofs remain unchanged if no participant needs to "jump forward" to any future round after GST, as the rebroadcast implies no messages of the current round are lost by the time GST is reached, not dropping any messages.

- ***Validity***: Validity is preserved by the fact that Lemma Consistent Proposal is preserved: a participant jumps forward to the beginning of a round only after terminating QUALITY (line 111), updating its set C to include a value v other than baseChain only if it receives a strong quorum of QUALITY, or a strong quorum of PREPAREs for v in some round as evidence in the received CONVERGE that justified the jump at line 106 (same conditions under which a participant already updated C by default).

- ***Agreement***: Agreement remains unchanged, as we show by contradiction. Suppose two correct participants p, p', that decided on different chains c, c'. This is only possible if they each gather a strong quorum of DECIDE messages for c, c', respectively. By quorum intersection and the fact that the adversary controls less than a weak quorum of participants, it is impossible to gather a strong quorum of DECIDE messages for c, c', as correct participants never jump once they reach the DECIDE phase, and only send one DECIDE message (perhaps rebroadcasting it). A contradiction.

- ***Termination***: We consider termination after GST and once Δ-synchrony is met. If one participant is at the DECIDE phase, then at the first (re)broadcast of its DECIDE message after GST (lines 63-84) all correct participants will deliver the DECIDE message immediately (lines 94-95), jump to the DECIDE phase (lines 53-56), and broadcast their own DECIDE, terminating with a decision for the value attached to the DECIDE phase.

We also show that it is impossible to receive two valid DECIDE messages with valid evidences

for different values v, v' such that v != v'. The proof is analogous to the proof of agreement (reaching a contradiction by quorum intersection): a valid DECIDE contains as evidence a strong quorum of COMMITs for the same round, and no correct participant sends conflicting COMMIT messages in the same round for two different values. It is also impossible to reach a strong quorum of COMMITs from different rounds for different values, thanks to the Lemma 3 (Byzantine carry-over decided proposal): all valid CONVERGEs, once a correct participant jumps to the DECIDE phase for a value v, contain v as proposed value (preserved when jumping forward to a future round by lines 102-109 and the fact that a round is executed in full, even if a correct participant reaches that round by jumping from a previous round, except in the cases when it jumps again executing again lines 102-109).

We have thus shown that if one participant is in the DECIDE phase, then after GST all participants terminate. We now show that if no participant is in the DECIDE phase, no participant is stuck and they all eventually reach a round in which they receive messages from all correct participants before the timeout of each phase.

To show that no participant is stuck, we need to show that, in the presence of scattered correct participants across rounds, they all eventually jump once. For participants to be at different rounds, there must be a fastest partition of correct participants that, together with Byzantines, makes a strong quorum (so that they progress to a future round). Before GST, it can be however that the fastest partition of correct splits in two, with one sub-partition P receiving all messages in a COMMIT phase, and going to one extra round r, while the other sub-partition Q remains stuck in the COMMIT from the previous round r-1. If no sub-partition between P and Q takes place, then all slower participants eventually jump to round r as there is a weak quorum of PREPARE for round r. If instead the sub-partition takes place then, after GST, then all participants slower than those in Q will then jump to round r-1 (as P and Q send PREPARE for round r-1, by lines 77, 82', and they make a weak quorum), reaching participants in Q. Since all correct participants not in P are then stuck at most in the COMMIT phase of round r-1, and participants in P are in round r, participants in P rebroadcast also their COMMIT for r-1 (line 81), allowing all correct participants to progress to round r.

Having shown that after GST all participants reach the same round, we show that they terminate in a future round after jumping. As shown in Lemma 2' (Once GST => one jump), after GST, eventually, the fastest participant sends its PREPARE for some round r, after the slowest participant starts the CONVERGE phase, with no more jumping by any participant other than to the DECIDE phase. Furthermore, the fastest participant can only proceed to the COMMIT phase of round r once a weak quorum of correct participants start the PREPARE phase of the same round. Since timeout(r+1) = 2*timeout(r), this weak quorum of faster participants receive all CONVERGE messages from each other in round r+1 on. It only suffices to prove that there will eventually be a round after which the timeout will be large enough for all participants to receive the CONVERGE messages from all other correct participants before terminating the CONVERGE phase. This occurs by the fact that there is eventually a round r'' > r such that all CONVERGEs are received by all correct participants in round r'' before any correct participant finishes the CONVERGE (since, after GST, if the fastest participants can skip the timeout of a

phase, so can the slowest participants, thanks to the modifications at lines 31, 39, and that timeouts double per round thanks to lines 50, 103). In the first round greater than r'' such that the winning ticket is held by the correct participant holding a proposal value *c* that is part of the set of candidates C for all correct participants (which, by Lemma Consistent Proposals, exists), termination is guaranteed: all correct will send and receive PREPAREs and COMMITs from each other for *c*, jump to the DECIDE phase, and send DECIDE for *c.*

QED

Time to Decision After GST.

In the following two theorems we analyze how likely it is to have a fast decision.

Theorem. (Decision after GST). After GST, all correct participants decide in O(n) rounds in expectation (in the worst case).

Proof. By Lemma Consistent Proposals, all proposals are prefixes of each other. We consider the proposals of correct participants represented in an array D. Let us consider D be a sorted array by the weight of the proposal, such that 2n/3 <= |D| <= n and, and let D[i] be the subarray of all proposals or suffixes of proposals whose proposers hold a quorum with weight i \in [0,1]. After GST, by Lemma 2' all correct participants receive all CONVERGE messages from other participants before they finish the same CONVERGE phase. Let r be the first round in which this occurs. The first round after r in which the winning ticket is proposed by a correct participant, such that the associated value is in D[2n/3], will terminate in a decision. This round is guaranteed to happen as there is at least one correct participant whose proposal is a prefix of all other proposals by correct participants.

Theorem. (Good case fast decision after GST). After GST, if there is a constant $a \in (0,1]$ of QAP held by correct participants that have as proposals (not necessarily the same proposal) values that are in the candidate set of all other correct participants, then correct participants decide in O(1) rounds in expectation (in the worst case).

Proof. The proof is analogous to the above theorem. With the exception that there is at least a weak quorum of correct participants such that decision is guaranteed after r in the first round in which any of these participants holds the winning ticket, if not before. As such, in expectation, a decision occurs in *1/a* rounds, i.e. O(1).

In practice most participants will share a similar candidate set (either a number of them execute QUALITY receiving a strong quorum of messages on time, or none, having baseChain as proposal). If O(n) in worst case is a concern, there are two possibilities to reduce the worst case to constant number of rounds, by either (i) defaulting to propose baseChain after a number of rounds if mayHaveStrongQuorum=False for all other values

(in all executed rounds of the protocol), or (ii) periodically rebroadcasting the QUALITY message and dynamically allowing participants to update their candidate set and proposal as new QUALITY messages arrive (even after having finished the QUALITY phase), also only updating their proposal if mayHaveStrongQuorum=False for all other values in all executed rounds of the protocol. This can be left as future work. In general, we recommend (i), which translates into not deciding on an instance that suffered from big discrepancies at the start of it due to lack of synchrony, as (ii) can facilitate the known FPL (censorship) attack that a >33% adversary can perform (Appendix B). Note that the performance impact from (i) should not be too impactful, as the following instance, starting already after GST in Δ-synchrony, will mean no jumping and likely finalize a prefix other than baseChain. Note however that the number of rounds selected for the default to baseChain implies a synchrony assumption in the regular case where there is no jumping.