```
1   send(message):
2     toSend.enqueue(message)
3
4   expect(predicat):
5     toSend.enqueue(predicate)
6
7   deliver():
8     block until received is not empty
9     return received.dequeue()
10
11  run():
12    while (not badlist[recipient]):
13      x := toSend.top()
14      if (x is a message):
15        sending.enqueue(x)
16        toSend.remove(x)
17
18  onReceive(message):
19    x := toSend.top()
20    if (x is not a predicate) then return
21    if (not x(message)) then return
22    toSend.remove(x)
23    received.enqueue(message)
```

Figure 8: The message queue

```
101 server-run():
102   while (not badlist[recipient]):
103     if sending is empty then
104       send "ping"
105     else
106       send all messages in sending
107     wait for send_delay
108     send_delay := min(2send_delay, 10 min)
109
110 onReceive(msg):
111   remove from sending all messages acknowledged by msg
112   send_delay := 2Γ
113   call higher-level onReceive(msg)
```

Figure 9: Message queue helper functions (server side)

```
201 onReceive(msg):
202   remove from sending all messages acknowledged by msg
203   send all messages in sending
204   if msg ≠ "ping" then call higher-level onReceive(msg)
```

Figure 10: Message queue helper functions (client side)

A Message Queue

A.1 Overview

The message queue implements, in an incentive-compatible manner, a reliable channel with a simple "you don't talk to me, I won't talk to you" policy. The pseudocode implementing this policy is shown in Figure 8.

The message queue is challenging because it is implemented on unreliable links, so nodes must resend messages—which is considered a cost. The implementation must make sure that no node can save cost by unilaterally deviating from the protocol.

A.2 Strawman protocol

Consider a TCP-like protocol in which messages are resent periodically until they are acknowledged. This protocol is reasonnable outside the BAR model, but it is not be incentive compatible: rational nodes can reduce their cost by unilaterally deviating from the protocol.

A rational node r may sometimes resend a message unnecessarily, because the previous send reached the recipient. To skip this cost, rational nodes can rely on the fact that their recipient resends its messages: The recipient's response will indicate whether r's message reached it or not. So r only sends its messages when it receives a message, instead of periodically. Clearly, if both correspondants deviate in this manner then no messages are exchanged.

The problem, at the root, is that some work is redundant: a rational node can shirk work because it knows that the other node's actions are enough to ensure progress.

A.3 Message queue protocol

Our protocol assigns different roles to the ends of the communication. Only one end, the *server*, is responsible for resending messages (the server could be the node with the lower I.P. address, or any other deterministic function). This solves the work shirking problem identified above by making all work necessary.

We assume that nodes believe that messages sent to non-Byzantine nodes either reach their destination by time Γ (e.g. the TCP timeout), or never do.

This assumption ensures that the server does not gain by waiting more than required in between sends, in the hope to receive a delayed answer from the other node.

Figure 8 presents the pseudocode for the message queue. The resending protocol is shown in Figures 9 (server) and 10 (client). As shown in lines 12 and 102, the message queue stops sending messages to a node b if it believes that b is Byzantine. One thing that is not shown explicitely in the pseudocode is the connection between sends and expects (lines 111 and 202): filling an *expect* bubble accounts as acknowledging some messages, but not necessarily the last message sent. The protocol itself must indicate which expects acknowledge which sends.

The protocol uses a $badlist$ array to indicate nodes that are believed to be Byzantine: there is nothing to be gained by sending to these nodes and therefore the message queue should not send to them (Lemma 10). There is no step in the message queue itself that puts nodes in the badlist, these come from higher levels of the protocol.

B Terminating Reliable Broadcast

The protocol is described in Section 5.1, and the pseudocode is in Figures 11 and 12. In terminating reliable broadcast, the sender goes through a three-phase commit to get all nodes to decide on its value. If nodes time out waiting for the last message for the sender they elect a new leader; the leader then continues where the sender left off. In our pseudocode the sender is process 0, although in fact the sender changes between instances of TRB.

The node that is the sender instantiates two threads, one that runs the *sender* function and the other that runs the *senderListen* function. Other nodes instantiate n threads for n turns. Each of these threads runs the *leader* function if that node is leader for that turn, or *nonleader* instead. Turn 0 is then started, which allows the code to proceed beyond line 47.

```
1   // sender protocol for processor i (i=0)
2   sender(i):
3     updateBadlist(badlist)
4     start senderListen(i) in parallel
5     prop := (proposal,badlist,untimely)
6     nv := (prop, i)
7     s := hash(nv)
8     send (agree, t, nv, ⊥) to all others
9     expect ⟨agree-ack, t, s⟩_j in response
10    wait for a quorum a⃗ of answers
11    send (write, 0, nv, a⃗) to all others
12    expect b_j = ⟨write-ack, t, max_pol⟩_j in response
13    accumulate responses into the set R until:
14      there exists b⃗ ⊆ R s.t. not failed(b⃗, t),
15      or |R| = n − f − 1 (then pick any quorum in R for b⃗)
16    send (show-quorum, t, b⃗) to all others
17    if not failed(b⃗, 0) then decide(nv, a⃗, b⃗)
18
19  // sender thread that listens to all turns>0
20  senderListen(i):
21    expect (report-decision, t, nv′, a⃗′, b⃗′) from all others
22    wait for one
23    decide(nv′, a⃗′, b⃗′)
24
25  // leader protocol for processor i on turn t (t>0, i=leaderForTurn(t))
26  leader(t, i):
27    pol_t := waitUntilElected(t, i)
28    if pol_t == "done" return
29    r⃗ := readOldValue(t, i, pol_t)
30    nv := latest(r⃗,t)
31    s := hash(nv)
32    send (agree, t, nv, r⃗) to all except sender
33    expect ⟨agree-ack, t, s⟩_j from all j
34    wait for a quorum a⃗ of answers
35    send (write, t, a⃗) to all except sender
36    expect b_j = ⟨write-ack, t, max_pol⟩_j from all j
37    accumulate responses into the set R until:
38      there exists b⃗ ⊆ R s.t. not failed(b⃗, t),
39      or |R| = n − f − 1 (then pick any quorum in R for b⃗)
40    send (show-quorum, t, b⃗) to all except sender
41    if failed(b⃗,t) then start next turn
42    else decide(nv,a⃗, b⃗)
43
44  // non−leader protocol for proc. i on turn t
45  nonleader(t, i):
46    l := leaderForTurn(t)
47    wait until started
48    start timer t: if it fires, then start the next turn
49      time−out for t is chooseTimeout(t, i)
50    if (turn>0) then
51      send ⟨set-turn, (t + 1)⟩_i to l
52      x := giveOldValue(t, i)
53      if x == "done" then stop timer t; return
54    expect (agree, t, nv, r⃗) from l ; wait for it
55    send ⟨agree-ack, t, hash(nv)⟩_i to l
56    expect (write, t, a⃗) from l ; wait for it
57    // a⃗ is a vector of agree−ack for turn t with the same s we got
58    // s == hash(nv)
59    // r⃗ contains a quorum of read−ack for turn t
60    // such that either nv is the latest value in r⃗,
61    // or r⃗ contains only ⊥ values
62    if t > m_val_t then
63      (m_val, m_val_t, m⃗_a) := (nv, t, a⃗)
64    send ⟨write-ack, t, max_pol⟩_i to l
65    expect (show-quorum, t, b⃗) from l; wait for it
66    // b⃗ contains a quorum of write−ack for turn t
67    stop timer t
68    if timer t fired more than avg_latency + window ago,
69      then untimely[l] := untimely[l] + 1
70    if failed(b⃗, t) then start next turn
71    else decide(m_val, a⃗, b⃗)
```

Figure 11: IC-BFT TRB, high level functions

```
101 waitUntilElected(t, i):
102   expect ⟨set-turn, t⟩_j from all other nodes j
103   wait until started
104   start timer t: if it fires, then start the next turn
105     time−out for t is chooseTimeout(t, i)
106   t_r := now() + avg_latency
107   for every node j from which we do not receive the expected
108   time−out message between t_r − window and t_r + window:
109       untimely[j] := untimely[j] + 1
110   wait until:
111     we receive a quorum pol_t of these messages:
112       stop timer t
113       return pol_t,
114     or i calls decide(nv, a⃗, b⃗):
115       send (show-decision, t, nv, a⃗, b⃗) to all
116       stop timer t
117       return "done"
118
119 readOldValue(t, i, pol_t):
120   send (read, t, pol_t) to all except the sender
121   expect r_j = ⟨read-ack, t, val, val_t, a⃗⟩_j
122 from all j other than the sender
123   wait for a quorum r⃗ of answers
124   return r⃗
125
126 giveOldValue(t, i):
127   expect (read, t, pol_t) or (show-decision, t, nv, a⃗, b⃗) from l
128   wait to receive it
129   if it's the latter then
130     untimely[l] := untimely[l] + decisionfee
131     decide(nv, a⃗, b⃗)
132     return "done"
133   if i is the sender then return "skipping"
134   max_pol := max(max_pol, pol_t)
135   send ⟨read-ack, t, m_val, m_val_t, m⃗_a⟩_i to l
136   return "go on"
137
138 failed(b⃗, t):
139   return true if b⃗ contains at least one
140   ⟨write-ack, t, mp⟩_j with mp.t ≠ t,
141   or false if all q have mp.t == t
142
143 latest(r⃗, t):
144   If all r_j ∈ r⃗ have r_j.val_t == ⊥ then
145     return (⊥,t)
146   else
147     return pad(r_j.val) for r_j ∈ r⃗ with the largest r_j.val_t
148
149 updateBadlist(badlist):
150   // this function is different for the "quasi−synchronous" version
151   if there is some node j that sent us a malformed message then
152   badlist[j] := true
153
154 leaderForTurn(t):
155   if (t==0) return 0
156   return (t mod (n−1)) + 1
157
158 computePenance(untimely):
159   return a buffer of size untimely * latefee
160
161 chooseTimeout(t, i):
162   if i is leader next turn, then return timeout * 2^t
163   else pick the value that maximizes the likelihood that the
164   time−out message will reach the next leader in the center of its
165   window (in the absence of other information, pick timeout * 2^t).
166
167 decide((v, t), a⃗, b⃗):
168   myPenance := computePenance(v.untimely[i])
169   for every node x ≠ 0:
170     send (myPenance) to x
171     penance := computePenance(v.untimely[x])
172     expect (penance) from x
173   if node i was sender then
174     for all j: untimely[j] := untimely[j] − v.untimely[j]
175   if some node x is in f + 1 nodes' badlist then badlist[x] := true
176   if this process is not the sender and has not sent report−decision
177   to the sender yet then send (report-decision, t, nv, a⃗, b⃗) to the sender
178   let the application know that we decided on value v
179
180 // should "node" be prevented from participating in the next instance?
181 hasBubble(i, node):
182   if decide(...) was not yet called then return true
183   if decide((v, t), a⃗, b⃗) was called then:
184     if i has an outstanding expect for node in any of the turns 0 through
185       t (included), then return true
186     if there exists some turn in which
187       i has sent a message to node, and
188       i has an outstanding expect from node
189     then return true
190     else return false
```

Figure 12: IC-BFT TRB, low level functions

The send and expect calls in the pseudocode all refer to sending through a message queue. Each turn has one message queue per other node. Within one instance of TRB they are linked together so that if there is a bubble against node r in turn t, then sends on turn $t + 1$'s message queue to r are delayed until the bubble is filled.

Turns are created as needed, to make sure that no message will be discarded because the turn that would have inserted the expect for that message is missing. To prevent this situation it is enough to make sure that whenever some turn t finishes (because the last instruction of that turn's *leader* or *nonleader* method returns), turns $t + 1$ through $t + n$ are created. Turns that are created place expects on their message queue, and then wait to be started: this is done explicitely in the time outs (lines 48 and 104) or when a leader failed to gather a successful quorum for its write (lines 41 and 70). A write is successful in turn t if it includes a vector $\vec{r}$ such that $failed(\vec{r}, t)$ (line 138)

returns false.

The asynchronous protocol uses quorums of size $n-f-1$, and quorums cannot include the protocol designated as sender for this instance of TRB.

C TRB Correctness

C.1 Proof technique

To prove that a protocol is IC-BFT for a given model of rational nodes' utility and beliefs, one must first prove that the protocol provides the desired safety and liveness properties under the assumption that all non-Byzantine nodes follow the protocol. Second, one must prove that it is in the best interest of all rational nodes to follow the protocol.

We start by proving correctness assuming that all non-Byzantine nodes follow the protocol.

C.2 Correctness assuming incentives

Here we assume that all non-Byzantine nodes follow the protocol.

Definition 1. *A value v is said to be* proposed *in turn t, if a leader sends a valid* `agree` *message in turn t with value v.*

Definition 2. *A value v is said to be* chosen *in turn t if there is a quorum Q such that all non-Byzantine nodes in Q answered the* `write` *message for v in turn t before receiving the* `read` *message from any later turn t'.*

Lemma 6. *If two non-Byzantine nodes satisfy the expect for a write message in turn t with values v and v' respectively, then $v == v'$.*

Proof. Each write message has the format $(\text{write}, t, nv, \vec{a}, \vec{r})$, where $\vec{a}$ consists of a quorum of answers of format $\langle \text{agree-ack}, t, s \rangle_j$, and s is the hash of the value v.

Since any two quorums intersect in a non-Byzantine node, and such a node sends only one *agree-ack* message in a particular turn, it follows that the same *agree-ack* message is used in the $\vec{a}$ value for the write of v and v'.

This requires that the hash for v and v' be the same. Under the secure hash assumption, it follows that $v == v'$. □

Lemma 7. *If a value has been chosen in turn t, then no other value can be proposed in turn t', $t' > t$.*

Proof. By contradiction. Let v be the value chosen in turn t, and $t' > t$ be the earliest turn after t in which some node proposed a different value v'.

If v has been chosen in turn t it follows that all non-Byzantine nodes in a quorum Q have received a write message for v, but have not received a read message from any later turn.

For a value to be proposed, it needs to contain agree-acks from a quorum of nodes. Non-Byzantine nodes will respond with an agree-ack only if the agree message is well formed, i.e. the value v' that is proposed is consistent with the vector $\vec{r}$ that has been sent (in particular, $latest(\vec{r}, t') == v'$ and every element of $\vec{r}$ is a valid message).

Vector $\vec{r}$ contains signed values from a quorum of nodes Q' and cannot be modified. Since Q and Q' intersect in at least one non-Byzantine node, and the non-Byzantine node will send the value v, it follows that there is at least one entry in $\vec{r}$ stating that value v was written in turn t.

Since entries in $\vec{r}$ include $\vec{a}$ in addition to the value and turn, all non-$\perp$ values in $\vec{r}$, even if they are from a Byzantine node, must have been proposed earlier.

Moreover, t' is the earliest turn after t to propose a value other than v. So there cannot be any proposed value $v" \neq v$ with a turn number $t" > t$ in $\vec{r}$ received in turn t'.

Value v from turn t is therefore the value in $\vec{r}$ with the highest turn number, and $latest(\vec{r}, t')$ will return v. Therefore the leader in turn t' must propose value v. □

Lemma 8. *A value v is chosen in turn t only if v was proposed in turn t.*

Proof. A non-Byzantine node accepts a `write` message only after it accepted the corresponding `agree` message. Since all quorums contain at least one non-Byzantine node, it follows that for v to be chosen at turn t it must have been proposed at turn t. □

Theorem 3 (Safety). *If some non-Byzantine node decides on a value v in turn t then no non-Byzantine node will decide on a value other than v.*

Proof. A node decides on a value v only after either seeing evidence that the value was chosen, either through a show-quorum message (lines 42 or 73), a show-decision message (line 129) or report-decision (line 22). The previous two lemmas indicate that at most one value may ever be chosen. □

Theorem 4 (Liveness). *Eventually every non-Byzantine node decides.*

Proof. Since the time-out delays increase exponentially, during the synchronous period there will be some turn after which every leader is guaranteed to have enough time to complete without being interrupted by another leader election. Consider the first such leader who is non-Byzantine. That leader will be able to write a consensus value without interference, and it will have gathered a quorum of acknowledgments ($\vec{b}$) that show that no other leader was elected before the end of the write. That information allows nodes to decide. Since the leader is non-Byzantine he sends it to all and all non-Byzantine nodes decide, and report to the sender if necessary. □

Theorem 5. *The protocol satisfies the conditions for TRB.*

Proof.
- Termination is guaranteed by Theorem 4.
- Agreement follows from Theorem 3 and Theorem 4

- Integrity is assured because a leader cannot propose any arbitrary value. The expect in line 56 is satisfied only if the proposed value has been written earlier, or is $\perp$. The fact that a leader cannot propose an arbitrary value hence follows by induction on the turn number t.
- In a period of synchrony, if the sender is non-Byzantine then no non-Byzantine node will time out on the sender because the time out values are larger than the known guaranteed delivery time Δ. It follows that the sender will be able to complete the turn and get all non-Byzantine nodes to deliver the message.

□

C.3 Equilibrium and incentive compatibility

Background We now show that the protocol represents an equilibrium point. More specifically, it represents a *Nash equilibrium*. We start by introducing this concept and relating it to our domain.

Nash Equilibira are a game theory concept. Game theory studies "games" among rational players. In one-shot games, for example, every player i (we shall call them nodes from here on) simultaneously picks some *strategy* σ_i. The rules of the game determine a *utility* u for each node, as a function of its strategy and the strategy of the other $n-1$ nodes. The utility for node i can be written as the function $u_i(\sigma_0, \ldots, \sigma_{n-1})$, which we abbreviate $u_i(\sigma_i, \sigma_{-i})$.

The Nash equilibrium is defined as follows [29]:

$$u_i(\sigma_i^*, \sigma_{-i}^*) \geq u_i(s_i, \sigma_{-i}^*) \text{ for all } s_i \in S_i$$

Where σ_i^* is the strategy proposed to node i, and S_i is the set of all deterministic strategies i can choose from.

To link these concepts to our domain, we observe that the strategy represents which actions the node will take in response to events it can observe. In other words, the strategy is the protocol that the node follows. A game-theoretic "game" is determined by a function that takes every node's strategy as input and outputs a resulting utility for each node. In our case, the input is which protocol each node follows and node's utilities are determined by the costs and benefits that the node experiences from running the protocol. We define the cost precisely later in this section. The two differences between our setting and the traditional phrasing of the Nash equilibrium is that, first, the utility can be influenced by network delays so that rational nodes must reason based on their expected utility. Second, Byzantine nodes may deviate arbitrarily from the protocol.

In a way similar to how an assignment of strategies to nodes can be said to be a Nash equilibrium for a given game if no player can improve its utility by unilaterally deviating from the assigned strategy, we say that a given protocol is a Nash equilibrium if no rational node can improve its expected utility by unilaterally deviating from the assigned protocol.

Proof technique To prove that the protocol is a Nash equilibrium, we show that it is in every node's best interest not to deviate from the proposed protocol under the assumption that all other non-Byzantine nodes follow the protocol.

Showing that something is in the best interest of a rational node is dependent on what the node considers in its interest, but also of the node's beliefs and knowledge. For example, a node that knows that a given node x is Byzantine will see no incentive to send messages to x, whereas one that does not know who is Byzantine must instead consider the expected utility of sending a message to x.

A rational node r evaluates its utility u for a strategy σ by computing its worst-case expected outcome. The worst case is computed over the choices of which nodes are Byzantine, and what Byzantine nodes do. The expectation is over network performance. The outcome then includes the costs: sending and receiving messages and computing signatures, and the benefits are: having their own proposal accepted. Node r also includes future effects of its actions, for example whether some node(s) now consider r to be Byzantine (by setting the corresponding entry in $badlist$ to $true$) or whether nodes will ignore r in the future (because the $hasBubble$ function returns $true$). A change that would prevent r from participating in future instances of TRB is considered to have infinite cost since it robs r from an infinite number of beneficial instances of TRB.

Our assumptions are presented in the System model, Section 3. In short, we assume that rational nodes gain a long-term benefit in participating, we assume that they consider the worst-case outcome of their actions, and we assume that if they observe that the protocol is a Nash equilibrium then they will follow the protocol.

The simplest deviations are those that do not modify the messages that a node sends. In our state machine protocol, no such deviation increases the utility. We must then examine every message that the node sends and show that there is no incentive to either (i) not send the message (ii) send the message with different contents, or (iii) send the message earlier or later than required. Also, we must show that nodes have no incentive to (iv) send any additional message.

Our protocol also imposes two requirements that must be met in an implementation: (a) The penance is larger than the benefit of sending a time-out message late, and (b) The $\perp$ answer (in read-ack) is at least as large as the largest allowed proposal value. (that size does not need to be constant, it may grow as the sender fails to propose values)

Theorem 6 (Incentive Compatibility). *No node has any unilateral incentive to deviate from the protocol.*

In order to show that no deviation is beneficial, we systematically explore all deviations. Table 1 maps each deviation to the lemma that shows that it is not beneficial. The concern of nodes sending additional messages is covered by Lemmas 9 and 10.

	not send	send different	diff. time
set-turn	Lemma 11	Lemma 16	Lemma 14
read	Lemma 11	Lemma 17	Lemma 26
read-ack	Lemma 11	Lemma 18	Lemma 30
agree	Lemma 11	Lemma 21	Lemma 27
agree-ack	Lemma 11	Lemma 19	Lemma 30
write	Lemma 11	Lemma 22	Lemma 28
write-ack	Lemma 11	Lemma 20	Lemma 30
show-quorum	Lemma 11	Lemma 23	Lemma 29
show-decision	Lemma 11	Lemma 24	Lemma 31
untimely	Lemma 11	Lemma 15	Lemma 15

Table 1: Map of deviation to lemma

Lemma 9. *Rational nodes only send a message m to node j if j expects that message.*

Proof. The queue protocol discards messages that are not expected. Therefore no rational node i would send an unexpected message to a non-Byzantine node because it has no benefit, but some costs (cost of sending the message, plus any signature in the message). Sending an unexpected messages to Byzantine nodes cannot improve their worst-case behavior (if anything, it may help them drive the system to an even less pleasant state). Therefore, no rational node sends an unexpected message to anyone, Byzantine or not. □

Lemma 10. *Once a rational node i knows that some other node j is Byzantine, i will not send any further message to j.*

Proof. If j is known to be Byzantine (for example because it was observed deviating from an incentive-compatible protocol), then sending messages to it does not affect the worst-case outcome. In particular, node j can always opt to ignore any message from i. Therefore, there is nothing to be gained from the expense of sending messages to j. □

Lemma 10 is a natural consequence from the fact that nodes are rational and that they believe that some nodes may be Byzantine. Naturally, in the worst case Byzantine nodes will not do something so foolish as letting themselves be identified.

Lemma 11. *If a rational node r knows that not sending some expected message m to non-Byzantine node s would cause the $hasBubble$ function in s to return $true$, then r has incentive to send the message.*

Proof. If $hasBubble(s, r)$ returns $true$ (indicating that s believes that r has not fulfilled all its obligations toward s), then s will not answer r's messages in futures instances of TRB. In the worst case (for r), all f Byzantine nodes will ignore r. In that case, when r is sender in a later instance i, it will not be able to gather the required $n - f - 1$ answers to its $agree$ message (since $f + 2$ nodes will not be included: the sender itself, s, and the f faulty nodes). As a result $hasBubble(x, r)$ will be true for all non-Byzantine nodes in instance i. From then on, node r will not be able to send its proposals to anyone: it is effectively excluded from the state machine. Node r would forgo participation in an infinite number of future beneficial instances of TRB: no finite benefit from not sending the message m may be worth this cost. Node r will therefore make sure to send all expected messages whose absence would cause $hasBubble$ to return true. □

Lemma 12. *No rational node r (r is not the sender) can ensure with certainty that $\perp$ will be delivered in a given instance of TRB.*

Proof. Nodes can influence the delivered value through their actions. However, if Byzantine nodes were to follow the protocol, then in a period of synchrony the sender will be able to communicate with a quorum of nodes and get its value delivered regardless of the actions of r (in particular if r does not send any message).

Therefore, rational node r cannot ensure with certainty that value $\perp$ will be delivered as the result of r's actions. □

Lemma 13. *Rational nodes other than the sender have to do the same amount of total work if in a given instance of TRB the decision is $\perp$ instead of the sender's value.*

Proof. If a sender's proposal is not accepted, then the sender will propose it again next time. Lemma 12 indicates that if a sender tries forever, the proposed value will be eventually delivered. The total amount of work, therefore, is the same (of course, the utility may be different because a different number of messages may be exchanged). □

Lemma 14. *There is nothing to be gained by sending the timeout message earlier or later than the protocol calls for.*

Proof. The protocol requires non-leader nodes to send the timeout message for turn t ("set-turn(t)") as soon as they believe that turn t started: either because timer $t - 1$ fired (i.e. a time-out) or because show-quorum for turn $t - 1$ failed. The leader in turn t never sends "set-turn(t)", and the sender never sends set-turn messages either.

Starting the next turn earlier (or later, as the case may be) may influence the outcome of TRB (toward either $\perp$ or the sender's value), but that has no effect on the amount of work that node r has to perform (Lemma 13).

All the messages for the current turn must be sent, so there is no other benefit from starting a turn earlier.

Delaying the start of turn t may save a node some effort, because it is possible that the delay allows turn $t - 1$ to receive (or compose, if the node is the leader) a successful show-quorum message, so that there is no need to send the set-turn message anymore.

However, the recipient of set-turn expects that message at a given time (and follows the protocol by hypothesis), so if the node sends "set-turn" late it increases its chance of missing the

window, thus raising the expected cost through the penance mechanism. By requirement (a), this expected cost is larger than the expected benefit from potentially not having to send "set-turn" and going through an extra turn (potentially with value $\perp$). □

Lemma 15. *Rational nodes have no incentive to omit or modify the untimely message.*

Proof. The untimely message (computed in lines 109 and 130, sent in line 5) is intended to inflict additional cost onto nodes that are believed to be untimely. If a rational node r omits this message, then its agree message is malformed (see Lemma 21). Modifying the contents of the message does not change its size, and the untimely message sent by node r does not impact node r (it impacts everyone else, as lines 169–172 show). Therefore, node r has no incentive to modify the untimely message. □

Lemma 16. *There is nothing to be gained by sending a set-turn message with the wrong contents.*

Proof. Since set-turn only contains a turn number and a signature, wrong contents would be equivalent to either sending twice to the message queue or sending a malformed message (Lemma 9), or sending set-turn early (Lemma 14). □

Lemma 17. *There is no incentive to lie in the read request.*

Proof. The format of the read request is entirely determined (line 127), the only freedom being in the specific choice of which quorum of entries in the POL are filled. Since all POL entries have the same size, all choices result in a POL of the same total size and hence the same cost. Since using a different valid POL has no impact on the protocol and does not reduce cost, there is no reason why a rational node would choose one quorum over another. □

Lemma 18. *There is no incentive to lie in the response to a read message.*

Proof. There are only two different possible answers to a read message: either the sender's value, or $\perp$. Since the sender's value is signed and nodes cannot forge signatures, the only possible lie is to answer $\perp$ when, in fact, one has received a value.

This lie increases the likelihood of $\perp$ being delivered instead of the sender's value, which has two consequences. First, it changes the amount of work that must be done in this turn. However, as we argue in Lemma 13, nodes other than the sender expect to have to do the same amount of work even if they try to increase the likelihood of $\perp$ being delivered. Second, it increases the size of the messages that must be sent because the $\perp$ answer has the same size (in bytes) as the longest allowed proposal (requirement b). Therefore there is no benefit to lying in response to a read message. □

Lemma 19. *There is no incentive to lie in the response to an agree message.*

Proof. The answer to agree is entirely determined by the agree message itself, so any deviation would be equivalent to not sending a message that the leader expects. Lemma 11 shows that there is no incentive to do that. □

Lemma 20. *There is no incentive for a rational node r to lie in the response to a write message.*

Proof. The only choice in the response is max_pol, the latest leader that the node has received a message from. Since these messages are signed, the only possible lie for a rational node is to reply with some POL it has received.

Since the size of the POL is constant, the only benefit of replying with an older POL is to influence the protocol. As we argued before (Lemma 13), only the sender has a stake in influencing the decision and the sender does not receive write messages.

Remains the possibility that answering with a different POL will influence the number of turns that the protocol takes to complete (that's a cost). Answering with the requester's POL instead of a later one (if we received one) means that there is some chance that the requester now thinks its proposal succeeded when, in fact, it failed. The potential benefit would be that if the requester succeeds, then there is no need to send a time-out message to the next leader. However, the fact that node r heard from the later leader means that it has already sent a time-out message to the later leader, therefore there is no incentive to lie in the response to the write message. □

Lemma 21. *There is no incentive to send incorrect data in the agree message.*

Proof. The agree message (sent in lines 8 and 32) include the turn number, proposal, and $\vec{r}$. Changing the turn number would be equivalent to not sending the agree message, which would result in $hasBubble$ returning $true$ (Lemma 11). The protocol does not restrict which proposal the sender can send, other than the condition that it must include the untimely vector. Lemma 15 argues that there is no incentive to send an incorrect untimely vector. Leaders that are not the sender have no choice in the proposal, as it is entirely determined by the contents of $\vec{r}$ and the $latest$ function. The vector $\vec{r}$ itself contains signed answers from other nodes, so it cannot be tampered with, other than choosing which answers to include in $\vec{r}$. These deviations are covered by Lemma 27. □

Lemma 22. *There is no incentive for r to send incorrect data in the write message.*

Proof. Sending a value that does not match the agreed-up hash would cause everyone to consider r Byzantine. The vectors $\vec{r}$ and $\vec{a}$ are both constant-size and cannot influence the protocol other than marking r as Byzantine, so there is no incentive to change them either. □

Lemma 23. *There is no incentive for r to send incorrect data in the show-quorum message.*

Proof. That message contains information signed by others, so it cannot be faked by r. □

Lemma 24. *There is no incentive for r to send incorrect data in the show-decision (or report-decision) message.*

Proof. Both messages have the same content. Their size is fixed, and nodes cannot lie about the decided value because they cannot forge signatures. The only deviation would be to use a different quorum for $\vec{r}$, but there is no benefit to that. □

Lemma 25. *There is no incentive for a rational leader r to send a message in its leader turn t before the protocol indicates turn t should start.*

Proof. It may prevent the previous leader from succeeding. Leaders have no stake in the outcome, so all that preventing the other from succeeding achieves is potentially cause more set-turn messages to be sent.

The sender cannot start early because the protocol says it should start immediately. □

Lemma 26. *There is no incentive for a rational leader r to wait for more than a quorum of time-out messages before starting its leader duty.*

Proof. That would allow the leader to go the show-decision route instead of the normal three phase commit. We use the penance mechanism to balance the costs ($decisionfee$, line 130). □

Lemma 27. *There is no incentive for a rational leader r to wait for more than a quorum of answers to its read message.*

Proof. Waiting for more answers may allow the leader to go from a situation in which it must propose $\perp$ (because none of the answers so far have seen the sender's value) to one in which it can propose the sender's value (because one of the answers includes it, cf. the $latest$ function in line 143)—or the other way around.

These two situations do not modify the expected number of turns for this instance of consensus. They are also identical in term of message size, because the leader must pad the proposal to maximum size, the same size as $\perp$. The difference between the two is which value is decided in the end, which may change how much work the leader must go through in this instance. However, as we argue in Lemma 13, this does not change the total amount of work. There is therefore no incentive for r to deviate from the protocol by waiting for more answers. □

Lemma 28. *There is no incentive for a rational leader r to wait for more than a quorum of answers to its agree message.*

Proof. Getting more answers cannot influence the outcome, so there is no incentive to wait for more. □

Lemma 29. *There is no incentive to wait for more answers to* `write`.

Proof. The protocol lets you wait for $n - f - 1$ answers, waiting for more may get you stuck. □

Lemma 30. *There is no incentive to answer late to either a read, agree or write message.*

Proof. The effect of a late reply to these requests is to potentially slow down the leader (or sender), increasing the risk that this instance of TRB lasts one more turn and potentially influencing the outcome.

Only the sender has a stake in the outcome, and it does not answer to these messages. Remains the possibility of adding a turn, which would cause the rational node to send more messages and therefore increase its cost. Rational nodes therefore have an incentive to respond to these queries immediately. □

Lemma 31. *There is no incentive to send the show-decision message late.*

Proof. Once a leader r knows that it must respond with the show-decision message, then further waiting has no impact on its cost: nothing can remove the requirement on r to send that message. The leader therefore has nothing to gain by delaying the answer. □

C.4 Enlightening examples

The protocol is Figure 11 distinguishes between the sender and the leader: the sender proposes a value and, if it is not timely, a new leader is elected. This distinction may seem unnecessary, but in fact it is important that the sender not be involved in steps where it may influence whether its value gets decided. This can occur in two places.

First, the sending of the "set-turn" messages. Suppose an execution in which the sender receives a POL from a later leader, and then a write for the value $\perp$, indicating that the new leader did not see any of the messages sent by the sender. The sender may then have an incentive to send its "set-turn" message early to elect a new leader, in the hope that the new leader may see one of the written values and will attempt to write the sender's value instead of $\perp$.

Second, the answer to "read" requests. In the same scenario as described above, if the sender receives a write for $\perp$ by leader 1 and then a read request from leader 2, then the sender would have an incentive to deviate from the protocol and send its own value instead, pretending it hasn't received the message from leader 1.

In order to avoid both scenarios, we allow the sender to try to write its value only once: it cannot be elected leader in later turns, and read messages are not sent to it. Since the read and write quorums must still intersect in at least one correct node and there must be a quorum of correct nodes among all nodes but the sender, it follows that $n \geq 3f + 2$.

D Replicated State Machine

The "hasBubble" function (Figure 12) is used to determine whether a given node should be allowed to participate in the next instance of TRB.

The replicated state machine provides the following guarantee under our liveness assumption that all non-Byzantine nodes get some overall benefit from participating in the state machine.

Theorem 7. *If non-Byzantine node a submits some command c to the state machine then eventually every non-Byzantine node n in the state machine will deliver c.*

Proof. Eventual synchrony guarantees that eventually a gets its turn as sender in the state machine. TRB's non-triviality condition then guarantees that a will successfully deliver its proposal. Once a is done with earlier submissions it will submit c, which it will deliver. The agreement condition guarantees that all non-Byzantine nodes will deliver c as well. □

E Work Assignment

This section addresses issues related to work assignment and relevant efficiency optimizations. In general, work assignment is used to reduce replication factors associated with running a protocol and to increase communication efficiency and reliability. The work assignment protocol leverages the state machine to replicate the assignment of work to a specific node or set of nodes. The work itself is then performed on the specific nodes. In general, the messages and execution of allocation are orders of magnitude less expensive than the execution of the work itself.

The work assignment protocol proceeds in 5 basic steps: (1) submit request to state machine, (2) state machine delivers request, (3) subset of state machine performs request, (4) result is sent back to all nodes in state machine, (5) all nodes in state machine forward request to the requester and done.

For all proofs in this section, we make the "sufficient benefit" assumption, that is rational node a gains sufficient benefit from membership to outweigh the cost of participating in the system if no more than f nodes deviate from the protocol.

Let w be work instructions, a, b be nodes in state machine A. Let u be the result of performing w. We also assume that all liveness conditions are met.

E.1 Work assignment to individual nodes

Here we address issues related to assigning work to an individual node. Let A be a state machine, a and b be nodes in the state machine, and w be work assigned by b to a. We assume that the result of performing w can only be acquired through actually performing w (i.e. that w is an unforgeable operation). The most relevant of this is the following lemma:

```
1   // protocol for execution of assign(w, a, b) on p ∈ A
2   execute(assign(w, a, b)):
3     if (p = a)
4       r := perform(w);
5       ∀c ∈ A c.send(r);
6
7     start w.timer;
8     a.expect < m : m.reqHash = hash(w) >;
9     m := a.deliver();
10    cancel w.timer;
11    b.send (witness(m, a));
12    if (p = b)
13      responses := {};
14      ∀c ∈ A start w_c.timer;
15      ∀c ∈ A
16        c.expect < m : m = witness(u : u.reqHash = hash(w)) >;
17        responses := responses ∪ c.deliver();
18        cancel w_c.timer;
19        if |responses| ≥ f + 1
20          process(responses);
21
22  \\ protocol for handling w.timer
23  on timeout(w.timer):
24    b.send(witness(''no evidence''));
25    badlist[a] := true;
26
27
28  \\ protocol for handling w_c : timer
29  on timeout(w_c : timer):
30    badlist[c] := true;
31
32  \\ protocol for processing responses
33  process(responses):
34    hand responses up to the caller of b
35
36
37  \\ protocol for execution of assign(w, B, b) on p ∈ A
38  execute(assign(w, B, b)):
39    if (p ∈ A ∩ B)
40      if (r not already submitted to B)
41        r = B.submit(w);
42      else
43        r = B.result(w);
44      ∀c ∈ A c.send(r);
45    ∀a ∈ A ∩ B
46      start w_a.timer;
47      a.expect < m : m.reqHash = hash(w) >;
48      m := a.deliver();
49      cancel w_a.timer;
50      b.send (witness(m, a));
51    if (p = b)
52      responses = {};
53      ∀c ∈ A start w_c.timer;
54      ∀c ∈ A
55        c.expect < m : m = witness(u : u.reqHash = hash(w)) >;
56        responses := responses ∪ c.deliver();
57        cancel w_c.timer;
58        if |responses| ≥ f + 1
59          process(responses);
```

Figure 13: Work assignment protocol

Lemma 32. *If state machine A executes the command* assign(w, a, b) *and a is rational, then a will perform w.*

Proof. Line 8 of Figure 13 introduces an expect to the message queue. If the expect is not filled, then a will be added to the $badlist$ in line 25, resulting in a's loss of access to the state machine and subesequent loss of benefit. Since w is a non-forgeable operation, a must perform w to have a result which will be accepted by the message queue. □

It is important to note that the exact semantics of "performing w" must be suitably defined by the application. In the context of PIB, the result of performing a store request is returning a Receipt or StoreReject for that request. There may be additional side effects (such as storing the actual chunk if the receipt is returned) which must be enforced at the application level.

E.2 Work assignment to another state machine

We can also partition the nodes of a system into multiple state machines. When this is done, it becomes necessary to assign work between state machines. The following lemma provides

certain guarantees when our state machine liveness criteria are met:

Let A and B be state machines which share $f+1$ nodes in their intersection.

Lemma 33. *If A executes* assign(w, B, a) *then B will execute w.*

Proof. By Theorem 7, if rational $b \in B$ submits w to B then B will execute w. It remains to show show that $\exists b$ that will submit the request to B. Line 47 inserts an expect into the message queue. By Lemma 11 rational b will send the expected message which is the result of w being performed by B. Since $|A \cap B| = f + 1$, $\exists b \in A \cap B$ such that b is non-Byzantine. If b is altruistic it submits w to B trivially. Let b be rational. Since b is rational, b assumes that the other f nodes in the intersection are Byzantine and will not submit w to B (thus possibly preventing b from returning the actual result of w to A). So in order to get a valid result of w, b must submit w to B. □

E.3 Credible threats

Ordinarily, all work is assigned through the state machine. A node submits an *assign* request to the state machine, the request is replicated on all nodes, and finally some subset of the nodes performs the request and the result is returned to the state machine. This can lead to problems if the requests themselves are large or costly to store and transport. A performance optimization is to instead submit a promise to assign work to the state machine, a direct message assigning the work to an individual node (submitting the request through the state machine only if the initiating node does not get a response back directly), and then report the result of the work assignment back to the state machine. The actual protocol is presented in Figure 14. There are two technical difficulties in implementing such a scheme: (1) should the requester actually submit the direct request and (2) should the requestee submit the direct request.

When requests are considered in isolation, the answer to both (1) and (2) is "no." Due to our modeling assumptions, in a pairwise communication rational nodes assume that the other node is Byzantine and plan for the worst action the other node could do to them. For both cases, the "worst" thing would be to require the state machine to be used regardless. Since this is a possibility, neither node will use the "fast" path solution. We address this by introducing *open* threats.

Definition 3 (Open Credible Threat). *A credible threat is* open *iff the requester can expect a response from a fast path request, based on the current time, before the requester must submit the request to the state machine in order to guarantee being able to fulfill his vow.*

Since rational nodes assume the maximum number of Byzantine nodes in the worst configuration, rational nodes will assume that they receive requests from Byzantine nodes, and that Byzantine nodes will send the request through the state machine regardless of whether or not the rational node responds directly. If, however, there are at least $f + 1$ nodes who have issued concurrent *open* credible threats against a rational node, at least one of them must be non-Byzantine so will in fact not go through the state machine if the node responds directly, and the rational node has reason to respond directly upon receiving the direct request. Similarly the sender of the request would expect to be ignored unless it currently has $f+1$ *open* credible threats issued. In order to quantify the necessary conditions for the fast path to be used, we must introduce the concept of *sufficient open threats*.

Definition 4 (Sufficient Open Threats). *There are sufficient open threats for a to follow the fast path iff $a \in A$ such that $|A| \geq f + 1$ and $\forall b \in A$, b is the requester or recipient of a credible threat from $B \subset A$ such that $|B| \geq f + 1$.*

Lemma 34. *Rational a will follow the fast path if there are* sufficient open threats.

Proof. If there are sufficient open threats, then other non-Byzantine nodes are assumed to use the fast path appropriately. By the *sufficient open threats* definition, there are open threats involving a from at least $f + 1$ distinct nodes. Let k be the number of distinct nodes involved in threats with a. Since $k \geq f + 1$, at least one of these nodes is non-Byzantine. The cost of following the fast path is the cost associated with sending a single message m. When following the slow path, the cost is at least nm (the cost of sending a single message to all other nodes in the state machine). So the expected cost of ignoring the fast path for all nodes is $(k)nm$ while the cost of following the fast path for all nodes i at most $(k)m+(f)(nm)$. If the node follows a fast path for only some of the nodes, then by our modeling assumption the ignored nodes will be assumed to be Byzantine, so no node should be ignored if the $k \geq f+1$ threshhold is met. So the expected cost is less if a follows the fast path and a will follow the fast path as suggested. □

While the *sufficient open threat* requirement is rather heavy handed, it is instructive to note that the requirements are structured so that the fast path will be used when the system is under heavy load – precisely when it using the fast path will have the most noticeable affect.

```
1   // protocol for submitting a vow to the state machine
2   // w is the request, a is the target, b is the requester
3   threaten(w, a, b):
4     RSM.submit(vow(hash(w), a, b));
5     delivered := false;
6     start (vow.timer1);
7     if (sufficient open threats)
8        a.send(w);
9        a.expect < m : m.reqHash = hash(w) >;
10       r := a.deliver();
11       delivered := true;
12       RSM.submit(r);
13       cancel(vow.timer1);
14
15
16  // process to execute a vow request
17  execute(vow(u, a, b):
18    if (p = a and sufficient open threats)
19       b.expect < m : hash(m) = u >
20       r := b.deliver();
21       b.send(r);
22    vowdelivered := false;
23    start (vow.timer2);
24    b.expect < m : m.reqHash = u >;
25    r := b.deliver();
26    vowdelivered := true;
27    cancel (vow.timer2);
28
29
30
31  on timeout(vow.timer1):
32   if (delivered = false)
33     r := RSM.submit(assign(w, a, b));
34     RSM.submit(r);
35
36
37  on timeout(vow.timer2):
38    if (vowdelievered = false)
39      badlist[b] := true;
```

Figure 14: Credible threat protocol