Chainlink Off-chain Reporting Protocol

Lorenz Breidenbach[*] Christian Cachin[†] Alex Coventry[*] Ari Juels[‡] Andrew Miller[§]

Version 1.2
24 February 2021

Contents

1 Introduction

This document describes the *Chainlink off-chain reporting protocol*, a new, more scalable, version of the protocol driving Chainlink's data feeds, realizing the vision for off-chain aggregation originally outlined in the Chainlink whitepaper [EJN17].

There are n *oracles* (or *nodes*) that monitor an off-chain data stream, typically an API reporting a price feed like ETH-USD. Periodically, the oracles jointly run the protocols outlined in this document (off-chain) to sign a report containing observations from many oracles. Once a report is produced successfully, one or multiple *transmitters* sampled from the oracle set transmit the report to a smart contract C running on a "main"

[*]Chainlink Labs

[†]The author is a faculty member at University of Bern. He co-authored this work in his separate capacity as an advisor to Chainlink Labs.

[‡]The author is a faculty member at Cornell Tech. He co-authored this work in his separate capacity as Chief Scientist at Chainlink Labs, in which he has a financial interest.

[§]The author is a faculty member at the University of Illinois at Urbana-Champaign. He co-authored this work in his separate capacity as an advisor to Chainlink Labs, in which he has a financial interest.

blockchain, which is considered to be Ethereum here, although no specific features of Ethereum are used by the off-chain reporting protocol. The contract validates the report, pays each oracle that contributed an observation to the report, and exposes the median of the reported values to consuming contracts on-chain. The first transmitter to successfully transmit the report to C is paid extra to make up for the Ethereum transaction fees she incurred during transmission of the report. Subsequent transmitters of the same report do not receive payment.

2 Design goals

At a high level, the design should achieve the following goals.

Resilience: The protocol should be resilient to different kinds of failures. Oracles may become Byzantine out of malice or due to buggy code. The chosen security model limits only the number of faulty oracles, not the types of faults.

Simplicity: We will implement and maintain this design, and wish to ship quickly to meet growing market demand, while supporting continual future improvements. We thus follow the *KISS* principle ("keep it simple, stupid") and try to make choices that lend themselves to straightforward implementation, reducing the likelihood of defects in the final system.

Low transaction fees: Communication between the oracles and computation performed by the oracles happens off-chain and is therefore (almost) free. In contrast, communication with C requires Ethereum transactions which carry hefty fees; for example, assuming a gas price of 30 GWei and an ETH price of USD 200 as of 2020, a transaction requiring 100k gas costs around 60 cents. In times of network congestion (e.g. caused by volatile markets), these fees could climb tenfold to USD 6 per transaction! We thus aim to minimize transaction fees, even if this results in protocol with higher off-chain computation and networking requirements.

Low latency: We want to minimize the time between the initiation of the signing protocol and the inclusion of the resulting transaction on the blockchain by C. Most end users of the data posted to C are DeFi trading platforms — they need fresh data. The performance of the data gathering protocol should only be limited by the network transmission latency (the network capacity seems of low relevance because the amount of transmitted data is small). Assuming real-world internet latencies, the protocol should produce a report within a few seconds. Typically, this time will be dominated by the time it takes to complete transmission of the resulting report to C. For example, in the case of Ethereum, we have to conservatively assume that inclusion of the transmission transaction may take on the order of minutes. However, on next-generation blockchains, inclusion may take less than second.

3 Model

Oracles. The system consists of n *oracles* $\mathcal{P} = \{p_1, \ldots, p_n\}$, which are also called *nodes*. The oracles may send messages to each other over a network and are identified by their network endpoints, i.e., a certificate on their cryptographic key material, which allows them to authenticate to each other.

The set of oracles is determined by an oracle smart contract C as introduced later. If it becomes necessary to change the oracle set that performs reporting, the oracle list in the smart contract is changed by its owner and the off-chain reporting protocol is restarted with the new set of oracles.

Each oracle p_i makes time-varying observations of a value such as a price. This is modeled as a value they can read at any time.

The protocol and its security analysis do not rely on any structure or distribution over these observations. The protocol only ensures that the on-chain reports include the real observations of sufficiently many correct parties. In most applications the idea is that these observations will be noisy versions of a real-world value, but this isn't necessary for the analysis here.

Failures. Any $f < n/3$ oracles may exhibit *Byzantine faults*, which means that they may behave arbitrarily and as if controlled by an imaginary adversary. All non-faulty oracles are called *honest* or *correct*. (Consequently, for $n \leq 3$, all oracles most be correct for the protocol to function correctly.)

For stating formal security guarantees, these faults may occur adaptively, where the adversary can choose the faulty nodes on the fly. Once faulty, a node remains faulty for the purpose of the model for the entire duration of the protocol. It is expected that the protocol operates usually with $n = 3f + 1$ since this gives optimal resilience.

In principle, the failure assumption covers also network failures and crashes, which may or may not be adversarially introduced. This assumption also means that no more than f nodes may become isolated from the network or crash. This is a weakness of the model, in that it does not cover, for instance, software errors which take down the whole network at once due to a malformed message. (In practice, we mitigate such errors through client diversity, for instance, by falling back to the previous *FluxMonitor* protocol that has been powering Chainlink's data feeds so far.)

To make our design more practical, however, we additionally permit *benign* faults in the following sense: A correct node may crash and become unresponsive for some time or it may become unreachable from some or all other correct nodes (as if during a network partition). When the node resumes operation after recovering, it restores some state from local persistent storage and will participate in the protocol again correctly. A benign fault is transient and must never influence the safety of the protocol.

With this refined model, which is only used informally, we want to achieve the following. If f oracles are Byzantine-faulty and c oracles are benign-faulty with $f < n/3$ but $f + c \geq n/3$ at any point in time during the protocol execution, then: (a) the protocol may lose liveness, but (b) always satisfies the safety properties.

For supporting this refined model and thereby increasing the resilience against crashes, some local state is always maintained on persistent storage. We will explicitly mention the variables for which this is the case.

Oracle smart contract and reports. The goal of the protocol is to repeatedly produce reports, which are recorded by the smart contract C on the Ethereum blockchain. A report is signed by sufficiently many oracles and consists of a sorted list with recent observations from a plurality of oracles. The report is submitted to C. The code of C, in turn, verifies the signatures, checks that the list of observations is sorted, records which oracles contributed, and stores the *median* of the observations as the reported value on the blockchain. The contributing oracles receive a payout.

Using the median among *more than* $2f$ observations ensures that the reported value is *plausible* in the sense that faulty oracles cannot move it outside the range of observations submitted by correct oracles.

Cryptographic primitives. The protocol uses public-key digital signatures and pseudo-random functions (PRF). The standard EdDSA and ECDSA schemes are used for digital signatures. A PRF may be implemented by HMAC-SHA256 or by Keccak256 (simply prepending the key to the message).

Timing model. We align the timing and network model with the partially synchronous model [DLS88] but make some simplifying choices.

To defend against (D)DoS attacks by outside entities, one may connect the oracles over a private network. This still leaves the system vulnerable to application-layer resource-exhaustion attacks by malicious oracles, e.g., exploiting a memory leak or flooding the network with irrelevant messages. Such issues are addressed by the implementation as much as possible.

Formally, partial synchrony means that the network is asynchronous and the clocks of the nodes are not synchronized up to some point called *global stabilization time (GST)*. After GST, the network behaves synchronously, the clocks behave synchronously, and all messages between correct nodes are delivered within some bounded delay Δ, and this remains so for the remainder of the protocol execution. In practice, a protocol may alternate multiple times between asynchronous and synchronous periods. Liveness is only ensured for periods of synchrony.

As a pragmatic choice and in contrast to the formal model, the maximal communication delay Δ is a constant configured into the protocol. A discussion of the timeout values used in the implementation is given in Section 6.

Network assumptions. The oracles may send point-to-point messages to each other over an unreliable network. All connections are authenticated and encrypted, that is, each oracle can authenticate every other oracle based on the list of oracles as determined by C on the blockchain.

Messages sent between correct oracles are not necessarily delivered in the same order in which they were sent.

It is possible for a network partition to temporarily isolate a large number of nodes. Such partitions can model node crashes and eventual reboots, as well. However, since nodes keep retrying messages, all messages among correct nodes get through *eventually*, once any impeding network asynchrony passes.

4 Overview

The overall goal is to periodically send *oracle reports* to the contract C, which runs on Ethereum. The Chainlink reporting protocol is structured into three algorithms, called *pacemaker*, *report generation*, and *transmission*.

Pacemaker. The pacemaker protocol drives the report generation process, which is structured into *epochs*. In each epoch, there is a designated *leader* that drives a *report generation protocol* for the epoch, similar to related protocols for consensus like PBFT [CL02]. However, the protocol described here does not ensure consensus and relies on C for resolving ambiguities that may occur due to transitions across epochs.

The pacemaker protocol runs continuously and periodically initiates a new epoch and a corresponding instance of the report generation protocol. Every report generation instance has a designated leader. The pacemaker protocol emits a *startepoch* event that triggers the leader to start the report generation protocol.

The pacemaker may also abort the currently running report generation instance if it does not observe enough progress. To this end, the pacemaker protocol receives *progress* events from the current report generation instance. Every oracle runs a timer with timeout duration Δ_{progress} that serves to watch the epoch leader's performance; every *progress* event resets that timer. When the timer expires, the oracle concludes that the current leader was not performing correctly and has not produced a valid report that may be sent to C. The oracle then moves to initiate a new epoch with another report generation protocol instance and a different leader. Neither the pacemaker protocol nor the report generation protocol actually sends out the report to C: the report generation protocol hands this task over to the transmission protocol.

The pacemaker protocol observes the progress only via the report generation instance through the *progress* events. Thus, Δ_{progress} need not depend on the worst-case transaction confirmation time on the blockchain.

The pacemaker protocol also responds to *changeleader* events originating from a report generation instance. This event indicates that the instance and its leader have run for the maximum permitted duration and that its epoch ends. When this occurs, the pacemaker also aborts the current report generation instance and starts the next epoch, with a new report-generation instance and possibly a different leader.

Report generation. Every report generation protocol instance corresponds to an epoch and has a unique identifier e and a leader node p_ℓ. A new epoch should be started whenever a sufficient number of oracles have determined that a new leader is needed. The report generation protocol of an epoch is structured into rounds. In each round, the protocol gathers *observations* and, if conditions for going forward are met, it generates a signed oracle *report*, which it hands over to the transmission protocol for transmission to C.

The rounds are controlled by the leader itself and by a timeout Δ_{round}. This timeout, which is only triggered by the leader, controls the frequency of rounds and of gathering observations. The value of Δ_{round} must be lower than Δ_{progress} and larger than the network latency required to complete a full iteration of the report generation protocol during periods of synchrony, plus a safety margin.

Once a sufficient number of observations are collected together into the more compact form of a report, the report is ready for being sent to C through the transmission protocol. The report contains enough signatures of the oracles for C to verify that the report is correct and valid. For preventing unnecessary reports, however, the leader may only generate such a report, and the oracles only participate in producing the report, under the following condition: either their view of the data-stream value has changed by more than a fraction α compared to the most recent value reported by C *or* more than an interval Δ_C time has passed since that report.

Transmission. The transmission protocol encapsulates the steps performed locally by each oracle for sending reports to C. Unlike the above two algorithms, the transmission protocol does not involve any communication among the oracles. The transmission protocol delays oracles pseudo-randomly to ensure a staged sending process, preventing that too many copies of the same report are sent off simultaneously to C in fault-free cases. This saves cost because C must process all reports it receives, also when it does not include a report in the blockchain.

5 Protocol description

We give a semi-formal description of the protocols using an event-based notation, as used in the standard literature [CGR11, Chap. 1]. A protocol is written in terms of a list of **upon**-handlers, which may respond to events or to conditions on the protocol's internal state. Handlers are executed *atomically*, i.e., in a serializable and mutually exclusive way, per protocol instance and per node such that no two handler executions of the same instance interleave.

A protocol instance (i.e., an instance of Algorithm 1, Algorithms 2– 4 or Algorithm 5 below) communicates with other instances running on the same oracle through *events*, which are triggered by

invoke event *example-event*(*arg1*, *arg2*) .

For each triggered event, a handler of the form

upon event *example-event*(*arg1*, *arg2*) **do**

is executed once. Events between two protocol instances executing on the same node are handled in the same order in which they were triggered (i.e., FIFO order). The execution is otherwise asynchronous, which means that the invoking protocol may only obtain output from an invoked protocol instance via further events.

Protocol instances communicate with peer instances running on different oracles by sending *messages* as

send message [EXAMPLE-MESSAGE, *arg1*, *arg2*] to p_j .

Messages also trigger events when they are received by the protocol on a destination node.

We make frequent use of broadcasts where an instance sends a message to all instances *including itself*:

send message [EXAMPLE-MESSAGE, *arg1*, *arg2*] to all $p_j \in \mathcal{P}$

Every correct node continuously resends every message until the destination node acknowledges their receipt; however, a correct node does not further retransmit any message once its view of the current epoch (in Algorithm 1) or round (in Algorithm 2–4) has been incremented from the one to which a message pertains. Every receiving node acknowledges every message it receives, even if a message is a duplicate.

We make use of timers throughout the protocol description. Timers are created in a stopped state. After being started, a timer times out once and then stops. A timer can be (re)started arbitrarily many times. Restarting an already running timer is the same as stopping it and then starting it afresh. Stopping a timer is idempotent.

5.1 Contract

The contract C is managed by an *owner* with administrative powers. The owner can set future payment amounts and add or remove oracles. When the owner modifies the oracle set, an Ethereum log event is emitted by the contract informing the oracles about the new configuration of the off-chain network. How the owner is instantiated is outside the scope of this document. In practice, trust-minimization considerations would suggest using a contract that limits the power of malicious actors, for example, by requiring multiple signatures by independent parties or enforcing a waiting period before changes can be enacted.

Contract C records the reports, each of which contains a logical timestamp and multiple observation values. When C receives a transmission with report R with a more recent logical timestamp than the maximum C has seen, C updates its record with the median of all observation values in the report and exposes this to consuming contracts. Transmissions with equal or smaller logical timestamp are ignored. Logical timestamps are implemented by epoch-round tuples (e, r), denoting the round r and the epoch and report-generation protocol instance e, in which the report was produced.

Upon successful transmission of R, contract C also pays each oracle that provided an observation to R and pays the oracle that submitted R to compensate it for its transaction fee expenditure.

We provide an Ethereum implementation of C, written in Solidity, in `OffchainAggregator.sol`.

Benchmarks. On the aforementioned Ethereum contract, we measure a transaction cost of ca. 291000 gas for 31 oracles for the first transmission of a given epoch and round. Any subsequent transmission with the same epoch-round pair is ignored (technically, it aborts using Ethereum's `revert`) and costs ca. 42000 gas.

5.2 Pacemaker

The pacemaker protocol in Algorithm 1 on p. 7 governs the choice of a series of leaders associated to epochs numbered in succession $0, 1, 2, \ldots$. For each epoch, one instance of the report generation protocol in Algorithm 2–4 is created and executed. If this does not produce a valid report after Δ_{progress} units of time, the oracle initiates a switch to the next epoch, and the corresponding next leader.

Description. Algorithm 1 is similar to the leader-detection algorithm specified by Cachin *et al.* [CGR11, Module 2.10, p. 61] and implemented by Algorithm 2.10 (p. 62) there. Many of the arguments made for the properties of that protocol carry over directly to this context. The algorithm is extended by means to tolerate lossy links, such that messages sent between correct oracles do not need to be buffered. Similar methods have recently been introduced in the literature [NK20; BCG20].

Every oracle p_i maintains a local variable e, which denotes the epoch in which p_i operates. Furthermore, a variable ℓ denotes the leader oracle p_ℓ of the current epoch, derived from e as $\ell \leftarrow \mathit{leader}(e)$. The variable *ne* tracks the highest epoch number the oracle has sent in a NEWEPOCH message. These variables are maintained on persistent storage.

The node broadcasts a NEWEPOCH message containing *ne* every Δ_{resend} seconds. This increases the probability that relevant NEWEPOCH messages get through, even if a message is dropped at some point. It also helps for integrating crashed oracles back into the protocol after they have recovered.

In Algorithm 1, the NEWEPOCH message plays the same role as the COMPLAINT message in Alg. 2.10 [CGR11]. Here, incorrect behavior by the current leader p_ℓ is determined by oracle p_i if p_i has not observed a valid final report by timeout Δ_{progress}.

If an oracle receives more than f messages of the form $[\text{NEWEPOCH}, e']$, each one containing some $e' > e$, it infers that at least one correct node wishes to progress to some epoch higher than e. The node chooses this epoch $\bar{e}$ as the $(f+1)$-highest entry of *newepoch* and sends out its own $[\text{NEWEPOCH}, \bar{e}]$ message. Since it is assumed that at most f nodes are Byzantine, receiving a message from more than f others implies that at least one correct node has earlier sent a $[\text{NEWEPOCH}, \bar{e}]$ message.

This protocol differs from Alg. 2.10 [CGR11] in that $\bar{e}$ is an arbitrary future epoch, not necessarily the next epoch from the receiving oracle's perspective. This allows the oracle to catch up if it misses messages pertaining to an entire epoch or more. Recall that correct nodes may also exhibit benign faults and be offline for some time. If f is close to $n/3$, this doesn't matter too much, since the protocol will not actually advance unless approximately $2/3$ of the nodes positively respond with NEWEPOCH messages. For smaller f, though, arbitrary delays could lead to multiple distinct perspectives on the current epoch.

The node continuously records the highest epoch numbers received from all others through NEWEPOCH messages. If a node observes more than $2f$ nodes wish to change to an epoch greater than e, the node aborts the currently running report generation protocol instance, and switches to epoch $\bar{e}$, where $\bar{e}$ is the $(2f+1)$-highest entry of *newepoch*. It also initializes the report generation protocol instance for $\bar{e}$. Because more than $2f$ indicated epoch $\bar{e}$ or greater, the node infers that more than f correct nodes wish to switch to a later epoch. This, in turn, implies that every correct node will receive more than f NEWEPOCH messages as well, since a correct node will send those message to all others. Hence, all correct nodes will eventually transmit a NEWEPOCH message containing epoch at least $\bar{e}$, according to the NEWEPOCH amplification rule, and initialize the report generation protocol for epoch $\bar{e}$.

Crashes and recoveries. At any time, some number c of nodes may exhibit benign faults and have crashed; they eventually will recover and resume operations (otherwise, they count as Byzantine). We reason about

Algorithm 1 Pacemaker protocol structured into epochs (oracle p_i).

state
$(e, \ell) \leftarrow (0, \mathit{leader}(0))$: current epoch and leader for p_i
$\mathit{ne} \leftarrow e$: highest epoch that p_i has initialized or for that it has sent a NEWEPOCH message
$\mathit{newepoch} \leftarrow [0]^n$: highest epoch received from p_j in a NEWEPOCH message
timer T_{progress} with timeout duration Δ_{progress} // leader must produce reports with this frequency, or be removed
timer T_{resend} with timeout duration Δ_{resend} // controls resending of NEWEPOCH messages

upon initialization **do**
initialize instance (e, ℓ) of report generation // see report generation protocol in Alg. 2–4
start timer T_{progress}

upon event *progress* **do** // the current leader is progressing with the report generation protocol (see Algorithm 2)
restart timer T_{progress}

function *send-newepoch*(e')
send message $[\text{NEWEPOCH}, e']$ to all $p_j \in \mathcal{P}$
$\mathit{ne} \leftarrow e'$
restart timer T_{resend}

upon event *timeout* from T_{resend} **do** // resend NEWEPOCH message every Δ_{resend} seconds, in case network drops it
send-newepoch(*ne*)

upon event *timeout* from T_{progress} **or event** *changeleader* **do** // leader is too slow, or tenure is over
stop timer T_{progress}
send-newepoch(*max*$\{e + 1, \mathit{ne}\}$)

upon receiving a message $[\text{NEWEPOCH}, e']$ from p_j **do**
$\mathit{newepoch}[j] \leftarrow \mathit{max}(e', \mathit{newepoch}[j])$

upon $\big|\{p_j \in \mathcal{P} \mid \mathit{newepoch}[j] > \mathit{ne}\}\big| > f$ **do** // NEWEPOCH amplification rule
$\bar{e} \leftarrow \mathit{max}\Big\{e' \Big| \big|\{p_j \in \mathcal{P} \mid \mathit{newepoch}[j] \geq e'\}\big| > f\Big\}$
send-newepoch(*max*$(\mathit{ne}, \bar{e})$)

upon $\big|\{p_j \in \mathcal{P} \mid \mathit{newepoch}[j] > e\}\big| > 2f$ **do** // agreement rule
$\bar{e} \leftarrow \mathit{max}\Big\{e' \Big| \big|\{p_j \in \mathcal{P} \mid \mathit{newepoch}[j] \geq e'\}\big| > 2f\Big\}$
abort instance (e, ℓ) of report generation
$(e, \ell) \leftarrow \big(\bar{e}, \mathit{leader}(\bar{e})\big)$
$\mathit{ne} \leftarrow \mathit{max}\{\mathit{ne}, e\}$
initialize instance (e, ℓ) of report generation // see report generation protocol in Alg. 2–4
restart timer T_{progress}
if $i = \ell$ **then**
invoke event *startepoch*(e, ℓ) // see report generation leader protocol in Alg.4

recovery from crashes solely in the context of Algorithm 1. A crashed oracle will not rejoin Algorithm 2–4.

In the following, assume that there are no simultaneous Byzantine faults and consider these scenarios:

$c \leq f$: When no more than f oracles have crashed, the protocol maintains liveness and progresses normally (based on the assumption that there are no further Byzantine faults). When an oracle crashes in some epoch *e*, it misses all messages sent until it recovers. Upon recovery, it will eventually receive more than f NEWEPOCH messages containing an epoch larger than *ne* (recall that *ne* is restored from persistent storage upon recovery). This oracle will then rejoin the protocol by sending a NEWEPOCH message itself, denoting an epoch larger than *ne*.

$c > f$: During the time when more than f oracles have crashed, the protocol loses liveness. The protocol will resume operations successfully once $n - f$ oracles are operating and more than f among them send a NEWEPOCH message with an epoch value of at least some e after recovery. This ensures that, eventually, more than f oracles send thier own NEWEPOCH message with an epoch of at least e and, in turn, all correct oracles announce epoch values $e' \geq e$. This implies that the correct oracles eventually start epoch at least e.

In the rest of this paragraph, we argue why the pacemaker protocol is able to resume after the crash and subsequent recovery of any number of correct nodes.

Observe first that the protocol ensures that for every correct oracle, the variable *ne* increases monotonically and, likewise, no entry in *newepoch* ever decreases. This follows directly from the protocol, e.g., for the assignments to *ne* that occur in the agreement rule, inside *send-newepoch*, from the way this function is called, and from the handling of received NEWEPOCH messages.

Furthermore, notice that *ne* $\geq e$ holds as well from the assignment to *ne* in the agreement rule and from the preceding reasoning. And since e is determined from the entries of *newepoch* that never decrease, also e increases monotonically.

Consider now a point in time when all correct nodes have recovered, there are Byzantine-faulty nodes, but the network timing has stabilized (i.e., a moment after GST). A stable situation also means that *when* a report generation protocol instance with a correct leader has been initialized by *all* correct oracles, this produces reports (and *progress* events) faster than Δ_{progress}; hence, no correct oracle times out on T_{progress} and initializes a further epoch like this.

In this situation, correct oracles that have recovered may have missed arbitrarily many messages. Hence, their locally highest epochs (stored in *ne*) may vary widely.

However, since all correct nodes resume the periodic transmission of NEWEPOCH messages, every correct node will soon receive $n - f$ NEWEPOCH messages and determine an epoch number $\bar{e}$ in the NEWEPOCH amplification rule that is reported by more than f nodes. Notice that $\bar{e} \geq$ *ne* for the local variable *ne* of at least one correct node p_j. However, it may be that $\bar{e}$ is larger than the highest epoch for which any correct node has initialized the report generation protocol.

Let p_s be some node with the $(f + 1)$-largest value of the *ne* variables among the $n - f$ correct nodes, and let e_s denote this epoch number. According to the protocol, $f + 1$ or more correct oracles will repeatedly send NEWEPOCH messages containing an epoch value of at least e_s. Since these messages are sent by correct oracles, every correct oracle will eventually have received at least $f + 1$ such NEWEPOCH messages and send a NEWEPOCH message with parameter e_s or higher as well.

This implies that every correct oracle eventually stores $n - f > 2f$ entries in *newepoch* that are at least e_s. Hence, every correct oracle has either already initialized report generation protocol instance e_s or will progress to epoch e_s and initialize the report generation protocol instance e_s.

It remains to show that no correct oracle has yet aborted the report generation protocol instance e_s. This follows easily, considering the monotonically increasing variables e and *ne* of each correct oracle: In order to progress to some epoch $e' > e_s$, a correct oracle would need more than $2f$ entries in *newepoch* containing e' or a higher value. Hence, accounting for f values reported by faulty oracles, more than f correct oracles would have sent NEWEPOCH messages containing e' or a larger value. However, the number of correct nodes whose *ne* variable may exceed e_s and that might actually have sent NEWEPOCH messages with parameter larger than e_s is at most f, according to the definition of e_s. This is a contradiction and shows that such an e' does not exist.

The *leader* function. The function *leader* : $\mathbb{N} \to \{1, \ldots, n\}$ maps epochs to leaders. It is important that it is balanced in the sense that for any long interval of epochs, every oracle becomes leader approximately equally often. It must be deterministic and computable by every oracle.

The standard implementation is to set

$$\mathit{leader}(e) = (e \bmod n) + 1.$$

The ordering of the oracles is determined by the list in C. If this order may be influenced by the oracles (for example, when ordered by their identifying public keys), this may provide an opportunity for a coalition of faulty oracles to arrange themselves consecutively, which could lead to long delays between correct operations of the protocol.

One may use a pseudo-random function for *leader* to avoid this risk. This has the advantage that the leader sequence remains unpredictable to any observer outside the set of oracles, ensuring that an external adversary cannot predict and attack the leader of a particular future epoch. This can be implemented with a cryptographic pseudo-random function $F_x : \{0,1\}^* \to \{0,1\}^k$, where x is a secret key (called the *seed*) and that maps an arbitrary-length input string to a fixed-length, k-bit output string that looks random to anyone who does not know the secret key. Interpreting k-bit strings as integers, one may then set

$$\mathit{leader}(e) = (F_x(e) \bmod n) + 1.$$

The seed is specified by the operator at startup of the protocol; it must be the same for all oracles.

The *leader* function may be adjusted so that no oracle is selected as leader in two consecutive epochs by changing the modulus to $n - 1$ and renumbering the oracles to exclude the current leader.

5.3 Report generation

Algorithm 2–4, on pp. 10 and 12 respectively, concern the current leader's coordination of the current round's oracle observations and signatures, assuming the leader is correct. All its messages are tagged implicitly by the epoch identifier (e, ℓ) and explicitly contain the round number (r). Notice this is one logical algorithm with two entirely separate parts, where Algorithm 2–3 contains code executed by all oracles including the leader p_ℓ and Algorithm 4 contains code executed only by p_ℓ. This means the rules on atomic invocation of clauses hold across the whole report generation algorithm.

All oracles start executing the protocol upon being initialized. The leader additionally receives a *startepoch* event, which actually triggers the message flow. The report generation protocol is only live when the leader is correct and may halt otherwise. The protocol executes a number of rounds and produces one report per round. After a predetermined number of rounds, it halts and signals to the pacemaker that it is time to kick off the next report generation protocol instance.

A new round is started every Δ_{round} units of time. The leader then sends first an OBSERVE-REQ message to all oracles. Oracles receiving an OBSERVE-REQ for round r from the leader move to round r, collect a new observation, sign it along with replay-prevention metadata, and send it back to the leader in an OBSERVE message.

Once the leader has $2f + 1$ valid OBSERVE messages for round r, the leader waits out a grace period of duration Δ_{grace} so that delayed observations may also be included in the report. (Oracles are paid for their observations, and we don't want to withhold payment from oracles that are *slightly* delayed, e.g. because the API they're querying is slow. This also provides an incentive for oracles to participate in a speedy manner.) When the grace period expires, the leader collates the observations, sorts them by observation value, and sends them to all oracles in a REPORT-REQ message. Recipients check whether the report should be assembled (as explained later), extract a compressed report containing just the observations and oracle identities, and send back a signature on that. With this signature, an oracle *validates* the report.

Once the leader has obtained more than f reports validated by signatures, it compresses them and produces an *attested* report for transmission to C. Taking more than f signatures ensures that the attested report has been validated by at least one correct oracle. Notice that the attested report is not unique, even when the leader is correct, since any $f + 1$ correctly signed observations form a valid attested report. Before the attested report is transmitted to C by Algorithm 5, the report generation protocol disseminates it to all oracles.

Algorithm 2 Report generation follower protocol instance (e, ℓ) (executed by every oracle p_i)

state

$r_f \leftarrow 0$: current round number within the epoch
sentecho $\leftarrow \perp$: echoed attested report which has been sent for this round
sentreport $\leftarrow$ FALSE: indicates if REPORT message has been sent for this round
completedround $\leftarrow$ FALSE: indicates if current round is finished
receivedecho $\leftarrow [\text{FALSE}]^n$: jth element true iff received FINAL-ECHO message with valid attested report from p_j

upon receiving message $[\text{OBSERVE-REQ}, r']$ from p_ℓ **s.t.** $r_f < r' \leq r_{\max} + 1$ **do**
  $r_f \leftarrow r'$ // oracle moves to next round as follower
  if $r_f > r_{\max}$ **then** // p_ℓ has exhausted its maximal number of rounds
    invoke event *changeleader* // see report pacemaker protocol in Alg. 1
    exit // oracle halts this report generation instance
  sentecho $\leftarrow \perp$; *sentreport* $\leftarrow$ FALSE; *completedround* $\leftarrow$ FALSE
  receivedecho $\leftarrow [\text{FALSE}]^n$
  $v \leftarrow$ current observation of the value oracle reports on (e.g. price)
  $\sigma \leftarrow \mathit{sign}_i([\text{OBSERVE}, e, r, v])$
  send message $[\text{OBSERVE}, r, v, \sigma]$ to p_ℓ

upon receiving message $[\text{REPORT-REQ}, r', R]$ from p_ℓ **s.t.** $r' = r_f \wedge \neg$*sentreport* $\wedge \neg$*completedround* **do**
  if verify that R is sorted with entries from $2f + 1$ distinct oracles
      $\wedge$ verify that all signatures in R are valid **then**
    if *should-report*(R) **then**
      $R' \leftarrow$ a compressed list of the form $[(w, k) \ldots]$ from $R = [(w, k, \sigma) \ldots]$
      $\tau \leftarrow \mathit{sign}_i([\text{REPORT}, e, r_f, R'])$ // validate the report
      sentreport $\leftarrow$ TRUE
      send message $[\text{REPORT}, r_f, R', \tau]$ to p_ℓ // the round continues
    else
      complete-round()

upon receiving message $[\text{FINAL}, r', O]$ from p_ℓ **s.t.** $r' = r_f \wedge$ *sentecho* $= \perp$ **do**
  if *verify-attested-report*(O) **then**
    sentecho $\leftarrow O$
    send message $[\text{FINAL-ECHO}, r, O]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{FINAL-ECHO}, r', O]$ from p_j **s.t.** $r' = r_f \wedge \neg$*receivedecho*$[j] \wedge \neg$*completedround* **do**
  if *verify-attested-report*(O) **then**
    receivedecho$[j] \leftarrow$ TRUE
    if *sentecho* $= \perp$ **then**
      sentecho $\leftarrow O$
      send message $[\text{FINAL-ECHO}, r, O]$ to all $p_j \in \mathcal{P}$

upon $\left|\{p_j \in \mathcal{P} \mid \mathit{receivedecho}[j] = \text{TRUE}\}\right| > f \wedge \neg$*completedround* **do**
  invoke *transmit*(*sentecho*) // see transmission protocol in Alg. 5
  complete-round()

Algorithm 3 Report generation follower protocol, helper functions, instance (e, ℓ) (executed by every p_i)

function *should-report*(R)
  // $C_O = (C_e, C_r, C_R, C_J, C_S)$ is the most recent report committed by C, as known to p_i
  // let t_C denote the time when C committed C_O
  $v \leftarrow$ *median*(C_R)
  $v' \leftarrow$ *median*(R)
  return $(C_e, C_r) = (0, 0) \vee (now - t_C > \Delta_C) \vee \left(\frac{|v-v'|}{|v|} > \alpha\right)$ // *now* denotes the current local time at p_i

function *complete-round*() // terminates the round in two cases ...
  completedround $\leftarrow$ TRUE // (1) when no transmit is needed and (2) when transmit has been invoked
  invoke event *progress* // see pacemaker protocol in Alg. 1; indicates leader is performing correctly
  // the round ends and p_i waits until p_ℓ initiates the next round

function *verify-attested-report*(O) // verify that more than f signatures attest report O
  $(\cdot, \cdot, R, J, T) \leftarrow O$
  for $k = 1, \ldots, f + 1$ **do**
    if $\neg$*verify*$_{J[k]}([\text{REPORT}, e, r_f, R], T[k])$ **then**
      return FALSE
  return TRUE

Algorithm 4 Report generation leader protocol instance (e, ℓ) (executed by p_ℓ)

state

$r_\ell \leftarrow 0$: current round number within the epoch
observe $\leftarrow [\perp]^n$: signed observations received in OBSERVE messages
report $\leftarrow [\perp]^n$: attested reports received in REPORT messages
timer T_{round} with timeout duration Δ_{round}, initially stopped
timer T_{grace} with timeout duration Δ_{grace}, initially stopped
phase $\leftarrow \perp$: denotes phase within round in $\{\text{OBSERVE}, \text{GRACE}, \text{REPORT}, \text{FINAL}\}$

upon event *startepoch*(e, ℓ) **do**
start-round()

upon *timeout* from T_{round} **do**
start-round()

function *start-round*()
$r_\ell \leftarrow r_\ell + 1$ // leader protocol moves to next round
observe $\leftarrow [\perp]^n$; *report* $\leftarrow [\perp]^n$
send message $[\text{OBSERVE-REQ}, r_\ell]$ to all $p_j \in \mathcal{P}$
start timer T_{round}
phase $\leftarrow$ OBSERVE

upon receiving message $[\text{OBSERVE}, r', w, \sigma]$ from p_j **s.t.** $r' = r_\ell \wedge i = \ell \wedge \textit{observe}[j] = \perp \wedge \textit{phase} \in \{\text{OBSERVE}, \text{GRACE}\}$ **do**
if *verify*$_j([\text{OBSERVE}, e, r_\ell, w], \sigma)$ **then**
observe$[j] \leftarrow (w, \sigma)$

upon $|\{p_j \in \mathcal{P} | \textit{observe}[j] \neq \perp\}| = 2f + 1 \wedge \textit{phase} = \text{OBSERVE}$ **do**
start timer T_{grace} // grace period for slow oracles
phase $\leftarrow$ GRACE

upon event *timeout* from $T_{\text{grace}} \wedge \textit{phase} = \text{GRACE}$ **do**
assemble report $R \leftarrow \texttt{sorted}((w, k, \sigma)\ \texttt{for}\ k, (w, \sigma)\ \texttt{in}\ \texttt{enumerate}(\textit{observe}))$
send message $[\text{REPORT-REQ}, r_\ell, R]$ to all $p_j \in \mathcal{P}$
phase $\leftarrow$ REPORT

upon receiving message $[\text{REPORT}, r', R, \tau]$ from p_j **s.t.** $r' = r_\ell \wedge i = \ell \wedge \textit{report}[j] = \perp \wedge \textit{phase} = \text{REPORT}$ **do**
if *verify*$_j([\text{REPORT}, e, r_\ell, R], \tau)$ **then**
report$[j] \leftarrow (R, \tau)$

upon exists R **s.t.** $|\{p_j \in \mathcal{P} | \textit{report}[j] = (R, \cdot)\}| > f \wedge \textit{phase} = \text{REPORT}$ **do**
$k \leftarrow 1$
for $j \in \{1, \ldots, n\}$ **s.t.** *report*$[j] = (R, \tau)$ **do** // collect the attestation of the final report
$J[k] \leftarrow j$; $T[k] \leftarrow \tau$
$k \leftarrow k + 1$
$O \leftarrow [e, r_\ell, R, J, T]$
send message $[\text{FINAL}, r_\ell, O]$ to all $p_j \in \mathcal{P}$
phase $\leftarrow$ FINAL

To start the dissemination, p_ℓ sends a FINAL message containing the attested report to all nodes. Once a node receives such a message with a properly attested report, it sends the report again to all nodes in a FINAL-ECHO message. Alternatively, when a node receives first a FINAL-ECHO message with a properly attested report and has not yet echoed one, then the node sends a FINAL-ECHO message with this report as well. Finally, when a node has obtained more than f valid attested reports, it invokes the transmission protocol through a *transmit* event and passes one of the attested reports to Algorithm 5. (Recall here that attested reports are not necessarily unique, as an incorrect leader could send different nodes attestations from different size-f subsets of the follower attestations it receives, or an incorrect follower could re-order the attestations it receives from the leader.) Notice that the amplification of FINAL-ECHO messages ensures that the transmission protocol is either started by all correct oracles or by none, but not necessarily with the same attested report.

The round may fail due to network delays or when the leader does not behave correctly. For instance, the leader may not gather enough signatures on the report in time or may fail to send some messages. The report generation protocol instance may then never produce an attested report or cause that only some of the oracles invoke the transmission protocol. Such liveness violations are caught by the calling pacemaker protocol, however, because no *progress* events are emitted.

The protocol prevents that a leader runs for more than $r_{\max}$ rounds, where $r_{\max}$ is a global parameter. This is to prevent a malicious leader from driving the protocol as quickly as possible and causing a denial-of-service attack, for instance through oracles exhausting their computational or network capacity, or by making oracles hit API limits. In particular, when the leader attempts to start any round numbered higher than $r_{\max}$, then every oracle exits the report generation protocol after receiving a corresponding OBSERVE-REQ message as a follower. The oracle signals this to the pacemaker protocol with a *changeleader* event and halts any further processing in this instance.

The function *should-report* determines whether the oracle should validate the current report and has a dual purpose. First, it prevents coalitions of up to f malicious oracles from spamming C with transmissions in order to quickly extract reward payouts for useless reports. Second, it enables running rounds in quick succession without spending lots of gas on spurious updates that don't materially change the median exposed by C.

Function *should-report* assesses if the oracle should participate in producing the report. As mentioned before, this is the case when the value to be reported has changed significantly with respect to value C_R committed by C most recently, enough time has passed since then, or when the report generator has just been initialized for the first time ever and the committed epoch-round tuple is still the initial value $(C_e, C_r) = (0, 0)$.

Since the function refers to the current time, it implicitly assumes clock synchronization. However, the duration Δ_C is a different order of magnitude than expected clock skew, so any split view on whether a report should be transmitted should resolve within an epoch or two. Note that asynchronous periods may also result in a split view among the correct oracles on whether the leader makes enough progress in the pacemaker protocol (Algorithm 1), i.e., when $f + 1$ oracles have faster clocks than the rest of the group including the leader.

A schematic execution of the report generation protocol is illustrated in Figure 1.

5.4 Transmission

Algorithm 5 is responsible for transmitting a final report O produced by Algorithm 2–4 to the smart contract C on the Ethereum blockchain. This means, it creates a suitable Ethereum transaction containing O. Under ideal conditions, the report generation algorithm starts the transmission protocol at roughly the same time across all oracles.

To avoid redundant transmissions and reduce gas costs, the algorithm first filters incoming reports. In particular, we aim to protect against a scenario where many rounds complete in quick succession and produce lots of similar reports. In such a case, we only want to transmit the first such report, and discard the following ones. To do so, we keep track of the latest incoming report L and only allow a report O through the filter if (1) C has seen a report at least as recent as L, i.e., there is no backlog of reports, or (2) if O's median observation value deviates sufficiently from the median in L.

After a report passes that filter, the algorithm proceeds in stages and is globally parameterized by a stage duration Δ_{stage} and a schedule $S = (s_1, \ldots, s_{|S|})$. In stage i, there are s_i distinct and randomly selected oracles that attempt to transmit the report to C. Each oracle is chosen at most once as a transmitter. Furthermore an oracle will transmit O only if the oracle has not yet observed the report on the blockchain, as committed by C. Stage i starts after duration $(i-1)\Delta_{\text{stage}}$ has elapsed. Writing $t_0 = 0$ and $t_k = \sum_{j=1}^{k} s_j$, this means that

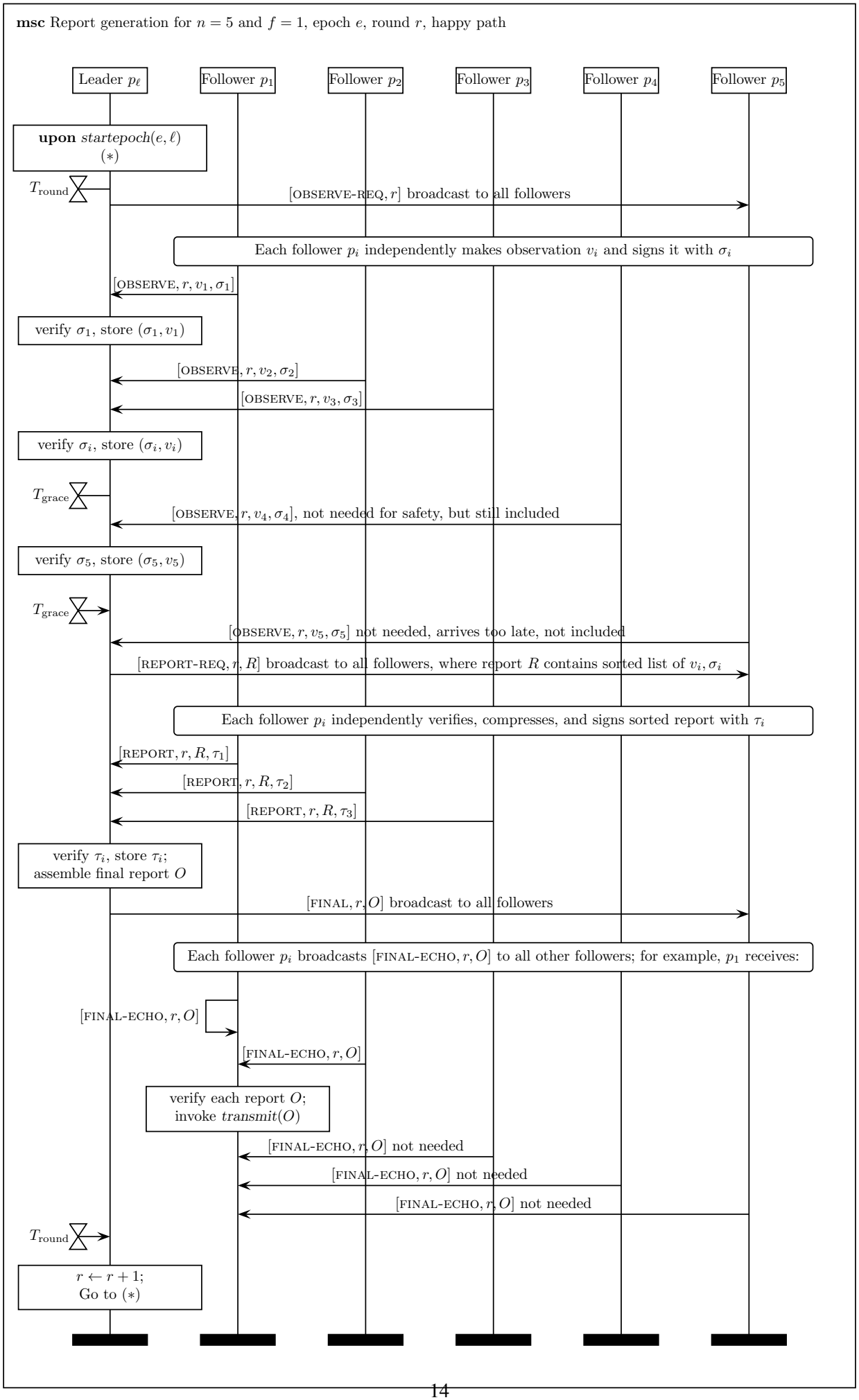

Figure 1. Sample execution and message trace of report generation with Algorithms 2–4.

Algorithm 5 Transmission protocol (oracle p_i).

state

timer T_{transmit}, initially stopped: delays until next report should be transmitted
reports $\leftarrow$ *heap-new()*: priority queue of time-report pairs (t, O), keyed on ascending time values
$L = (L_e, L_r, L_R, L_J, L_S) \leftarrow \perp$: latest report accepted for transmission
$C_O = (C_e, C_r, C_R, C_J, C_S)$: most recent report committed by C, as known to p_i

upon event *transmit*(O) **do**
$(O_e, O_r, O_R, \cdot, \cdot) \leftarrow O$
if $(O_e, O_r) \leq (C_e, C_r)$ **then** // O is outdated compared to C_O
return
if $L \neq \perp \wedge (O_e, O_r) \leq (L_e, L_r)$ **then** // O is outdated compared to L
return
if $L = \perp \vee \frac{|\textit{median}(O_R) - \textit{median}(L_R)|}{|\textit{median}(L_R)|} > \alpha \vee (L_e, L_r) \leq (C_e, C_r)$ **then**
$L \leftarrow O$
$\Delta_{\text{transmit}} \leftarrow$ *transmit-delay*(i, O_e, O_r)
insert $(\textit{now} + \Delta_{\text{transmit}}, O)$ into *reports* // recall *reports* contains (t, O) pairs sorted by ascending t
$(t, O) =$ *heap-peek*(*reports*)
restart T_{transmit} with delay $(t - \textit{now})$

upon event timeout from T_{transmit} **do**
if $|\textit{reports}| = 0$ **then**
return
$(\cdot, O) \leftarrow$ *heap-pop*(*reports*)
$(O_e, O_r, \cdot, \cdot, \cdot) \leftarrow O$
if $(O_e, O_r) > (C_e, C_r)$ **then**
send Ethereum transaction with O to C
if $|\textit{reports}| > 0$ **then**
$(t, \cdot) \leftarrow$ *heap-peek*(*reports*)
restart T_{transmit} with delay $(t - \textit{now})$

function *transmit-delay*(i, e, r)
$\pi \leftarrow G_x(e \| r)$ // derive pseudo-random permutation over $\{1,\ldots,n\}$
$k \leftarrow k$ such that $\sum_{j=1}^{k-1} s_j < \pi(i) \leq \sum_{j=1}^{k} s_j$ // assuming $s_0 = 0$
return $k \cdot \Delta_{\text{stage}}$

in stage k, the oracles on position $t_{k-1}+1, \ldots, t_{k-1}+s_k$ of π are supposed to transmit. By requiring that $\sum_i s_i > f$, we can ensure that there is at least one correct node that will transmit to C. The timeouts are such that reports to be transmitted from multiple rounds may overlap substantially, i.e., a report form the subsequent round may arrive before many oracles had a chance to transmit. A realistic setting is $n \cdot \Delta_{\text{stage}} \gg \Delta_{\text{round}}$ (see Section 6).

For the transmission protocol to achieve its goal, we require that a correct oracle will always be able to get a transaction included in the Ethereum blockchain within Δ_{stage} time. This assumes that (1) miners are actively mining the Ethereum chain and including transactions from their mempools according to the usual gas price auction dynamics and that (2) the oracle appropriately sets (or escalates) its gas price bid to have the transmission transaction included in the blockchain. These assumptions may be violated in practice, e.g., during times of particularly egregious congestions of the Ethereum chain. In such cases, transmission transactions will still be included eventually, but later and at a higher total gas cost than modeled here.

The selection of transmitting oracles occurs with a pseudo-random function $G_x : \{0,1\}^* \to \mathrm{Sym}(n)$, where x is a secret key known only to oracles and $\mathrm{Sym}(n)$ is the set of permutations of $\{1, \ldots, n\}$. Given the (implicit) protocol identifier, epoch, and round, G_x deterministically derives a permutation of the node set. As with the *leader* function described in Section 5.2, the secret x should not be known to the oracles before they are committed to their indexing $\{p_1, \ldots, p_n\}$, so that a malicious coalition cannot arrange themselves to dominate the early parts of the schedule.

Mechanism design. Faulty oracles may misbehave and ignore the global transmission schedule given by the algorithm, e.g., because their clock is malfunctioning or because they wish to earn the higher payment afforded to the first transmitting oracle. (Oracles that observe, but do not transmit, are paid less.)

The defense against this behavior is purely economical. The owner should monitor C and check that oracles follow the transmission schedule. We expect the owner to remove any oracles that consistently misbehave (i.e., submit too late or too early) from the protocol, preventing them from earning future reporting and transmission fees. Since such an attack threatens neither key safety nor liveness properties, and we expect the ratio between earnings from ongoing protocol participation to earnings from such an attack (prior to discovery) to be high, this defense should suffice in practice.

6 Implementation concerns

Notice that all protocols are functional under partial synchrony only to the extent that the timing characteristics of the network after GST are reflected by the chosen constants. Since they are fixed and not determined adaptively with respect to an unknown Δ, the protocol does not formally adhere to partial synchrony.

Example values for constants. We provide rough estimates of how we're planning to set many of the constants used in the protocol description for deployments targeting Ethereum.

Δ is the upper bound on communication latency during periods of synchrony. The choice of all other time constants is constrained by this. Estimate: 1 second

$\Delta_{\textbf{progress}}$ is the duration in which a leader must achieve progress or be replaced. Estimate: 30 seconds

$\Delta_{\textbf{resend}}$ is the interval at which nodes resend NEWEPOCH messages. Estimate: 15 seconds

$\Delta_{\textbf{round}}$ is the duration after which a new round is started. Estimate: 15 seconds

$\Delta_{\textbf{grace}}$ is the duration of the grace period during which delayed oracles can still submit observations. Estimate: 2 seconds

Δ_C limits how often updates are transmitted to the contract as long as the median isn't changing by more then α. Estimate: 5 minutes – 24 hours

α allows larger changes of the median to be reported immediately, bypassing Δ_C. Estimate: 0.2%-5% (depending on the underlying data feed)

$\Delta_{\textbf{stage}}$ is used to stagger stages of the transmission protocol. Multiple Ethereum blocks must be mineable in this period. Estimate: 30 seconds – 60 seconds

$r_{\textbf{max}}$ is the maximum number of rounds in an epoch. Estimate: 3 – 10

n is the number of oracles. In our initial implementation, we assume $n \leq 31$.

Domain separators. All hash and signature computations have to use proper domain separators, which we omit from the above protocol description for notational clarity. Domain separators include:

- Protocol identifier
- Address of C
- A counter of protocol instances (maintained by C)
- Chain identifier (e.g. for Ethereum mainnet)
- Set of oracles $\mathcal{P}$

7 Future work

In the current protocol, Δ_{round} controls the frequency at which a correct leader should create reports. It might be desirable to produce more frequent reports upon observing significant changes of the reported data stream, but will require careful design because it potentially opens us to price-manipulation attacks, and since we're considering Δ_{round} values much smaller than the average block interval, it's not clear how these reports would be represented on-chain.

We plan to adopt some form of aggregate or threshold signature scheme instead of the multisignatures used so far. The constant (in n) gas costs of signature verification would enable us to support larger oracle sets without major increases in on-chain cost.

As another extension for a later version, a new oracle list signed by a quorum of oracles from the current set could be used to change the off-chain oracle set without requiring the owner to intervene.

Finally, we plan to extend the protocol to support generalized aggregation functions rather than just the median, enabling advanced use cases such as *Fair Sequencing Services* [JBT20].

8 Analysis

This section contains a detailed analysis of the algorithms introduced earlier, consisting of definitions and semi-formal arguments of correctness.

8.1 Pacemaker

The pacemaker protocol (Section 5.2) runs continuously. It interacts with its environment by consuming (input) events of type *progress* and *changeleader*, by initializing and aborting report generation protocol instances, each identified by a unique (e, ℓ) pair, and by emitting *startepoch*(e, ℓ) (output) events at the leader.

***initialize* a report generation instance (e, ℓ):** Indicates that the oracle moves to epoch e and runs the corresponding report generation instance with oracle ℓ as leader. At most one report generation protocol instance is initialized for a particular (e, ℓ).

***abort* a report generation instance (e, ℓ):** Stops the running protocol instance identified by epoch e and leader ℓ.

***startepoch*(e, ℓ):** Indicates at oracle p_ℓ that the oracle should start epoch e as leader. This occurs only after the report generation instance (e, ℓ) has been initialized by p_ℓ.

***progress*:** Originates from the current report generation protocol instance and indicates that the leader has made adequate progress, according to the oracle's local view and clock.

***changeleader*:** Originates also from the current report generation protocol instance and indicates that this instance has reached its maximal number of rounds; therefore, the oracle should advance to the next epoch with a new leader.

Furthermore, the pacemaker protocol internally uses a timer T_{progress} with duration Δ_{progress} that captures the assumption that every report generation protocol instance should repeatedly emit *progress* or *changeleader* events that are no more than Δ_{progress} apart. Since the report generation protocol is driven by a leader oracle p_ℓ, we say that the *oracle times out on* p_ℓ when no *progress* or *changeleader* event occurs between two successive timeouts from T_{progress}.

In the following, we discuss liveness conditions only for stable periods (i.e., after GST).

Definition 1. *A* pacemaker *protocol has the following properties:*

Eventual succession: *If more than f correct oracles have initialized some epoch e with leader p_ℓ during a period of network synchrony after GST and subsequently either time out on p_ℓ or receive a changeleader event, then all correct oracles will advance to epoch $e+1$ after at most $2\Delta_{resend} + 2\Delta + 4\epsilon$.*

Putsch resistance: *A correct oracle does not abort the current epoch and initialize a new epoch unless at least one correct oracle has timed out on the leader of the current epoch or the report generation protocol has indicated to change the leader.*

Eventual agreement: *Eventually, every correct oracle has initialized the same epoch e with leader p_ℓ and does not abort it unless some correct oracle times out or indicates a leader change.*

We now show that Algorithm 1 implements a pacemaker protocol. We proceed by formulating and proving a number of lemmas. The first one characterizes how the correct oracles progress to subsequent epochs. Notice this holds also for asynchronous periods.

Lemma 1. *Suppose the maximal epoch that has been initialized by any correct oracle is $\bar{e}$ and let $\overline{ne}$ denote the maximal value of variable ne at any correct oracle. Then,*

(a) if no correct oracle times out and epoch $\bar{e}$ does not indicate a leader change at any correct oracle, then no correct oracle broadcasts $[\text{NEWEPOCH}, e']$ with $e' > \overline{ne}$;

(b) $\bar{e} \leq \overline{ne} \leq \bar{e} + 1$.

Proof. Recall the role of variable *ne* in Algorithm 1, which records the highest epoch for which the oracle has ever broadcast a NEWEPOCH message, and that *ne* is at least the current epoch value e for correct oracles.

To prove claim (a), note that an oracle with current epoch $\bar{e}$ can only advance *ne* to $\bar{e}+1$ independently, i.e., without receiving any NEWEPOCH messages, through the "leader is too slow, or tenure is over" handler. However, this is ruled out by the assumption of the lemma.

Hence, the only way for a correct node to increase *ne* beyond $\overline{ne}$ is via the "NEWEPOCH amplification rule." This triggers when the oracle receives $[\text{NEWEPOCH}, e']$ messages with $e' > \overline{ne}$ from more than f oracles.

We now establish an infinite descent to obtain a contradiction. Suppose that a correct oracle has current epoch $\bar{e}$ and its "NEWEPOCH amplification rule" is triggered, so that *ne* becomes greater than $\overline{ne}$. Since there are at most f faulty oracles, but more than f $[\text{NEWEPOCH}, e']$ messages with $e' > \overline{ne}$ have been received, at least one *other* correct oracle has sent $[\text{NEWEPOCH}, e']$. Hence this oracle's *ne* variable is *also* greater than $\overline{ne}$. This leads us back to the supposition at the start of this paragraph, with a different correct oracle. We continue this argument by induction and observe that $ne > \overline{ne}$ implies there are infinitely many distinct, correct oracles. However, the set of oracles is finite and $ne \leq \overline{ne}$ for all of them. Therefore, all correct oracles will correspondingly broadcast $[\text{NEWEPOCH}, e']$ with $e' \leq \overline{ne}$.

To establish claim (b), suppose towards a contradiction that $\overline{ne} \geq \bar{e}+2$. It follows from the above argument that at least one correct oracle has sent a NEWEPOCH message containing epoch $\bar{e}+2$ or higher, through the "NEWEPOCH amplification rule." However, this oracle has set *ne* to $e+1$ from its variable e according to the algorithm. But $\bar{e}$ is the maximum epoch value e of any correct oracle and therefore $e+1 = ne = \overline{ne} \geq \bar{e}+2 \geq e+2$, a contradiction. It follows that $\overline{ne}$ is at most $\bar{e}+1$. ■

The partially synchronous system model postulates a global stabilization time (GST), after which no more crashes occur, clocks are synchronized, and every message between two correct oracles is delivered within Δ. Suppose that the local processing time of any correct oracle after receiving messages is at most ϵ; this includes receiving multiple messages concurrently, i.e., ϵ is an upper bound on the processing latency of all messages and local events between receiving and sending messages over the network. This value is negligible compared to the other durations under consideration. The next lemma establishes a condition for reaching agreement after GST, in the sense that all correct oracles initialize the same epoch and remain in this epoch long enough.

Lemma 2 (Agreement)**.** *Consider an epoch e such that no correct oracle has initialized any higher epoch than e. If some correct oracle has initialized e before GST, then define a point in time τ to be GST; otherwise, let τ be the time when the first correct oracle initializes e. Let $\overline{ne}$ denote the maximal value of variable ne at any correct oracle at time τ.*

If $e = \overline{ne}$, no correct oracle times out, and epoch e does not indicate a leader change at any correct oracle during at least $\Delta_{agree} = 2\Delta_{resend} + 2\Delta + 4\epsilon$ after τ, then all correct oracles have initialized epoch e at time $\tau + \Delta_{agree}$ and do not abort it afterwards unless epoch e indicates a leader change.

Proof. Notice that the local epochs and the highest epochs (stored in e and *ne*, respectively) of the correct oracles may differ because of crashes. In particular, correct oracles that have recovered may have missed arbitrarily many messages.

Let p_i be the oracle that has initialized epoch e according to the assumption. It has received more than $2f$ NEWEPOCH messages containing an epoch value of at least e. More than f of those messages were sent by correct oracles. Even if these nodes had crashed, they meanwhile recovered, restored their e and *ne* variables to their highest values before the crash, and operate correctly from now on. Thus, more than f correct nodes retransmit NEWEPOCH messages with an epoch value of at least e at most $\Delta_{\text{resend}} + \epsilon$ after τ.

Thus, after at most $\Delta_{\text{resend}} + \Delta + 2\epsilon$, every correct oracle has received and processed more than f NEWEPOCH messages containing an epoch value of at least e. According to the algorithm and the "NEWEPOCH amplification rule," since epoch values of at least e are reported more than f times, every correct oracle broadcasts a NEWEPOCH message containing epoch e or higher after no more than $2\Delta_{\text{resend}} + \Delta + 3\epsilon$.

These messages are received by all correct oracles after at most $2\Delta_{\text{resend}} + 2\Delta + 4\epsilon = \Delta_{\text{agree}}$. Hence, every correct oracle has received $n - f > 2f$ NEWEPOCH messages with epoch at least e and has initialized epoch e after Δ_{agree}. Lemma 1(a) implies that no oracle initializes a higher epoch afterwards, until epoch e indicates a leader change. ■

Notice that the previous lemma only bounds the time after GST for reaching agreement on a particular epoch e when $e = \overline{ne}$. If some correct nodes have initialized epoch e at GST and $\overline{ne} = e + 1$, which is the only alternative to this condition according to Lemma 1, it means that that variable *ne* of at least one correct oracle is equal to $\overline{ne}$. The situation may remain like this for an unbounded period. But once the faulty oracles cause a correct oracle to initialize epoch $\overline{ne}$ by sending NEWEPOCH messages with parameters at least $\overline{ne}$, the condition of the lemma applies again and bounds the time that the correct oracles need for progressing to epoch $\overline{ne}$. We can summarize this behavior less precisely in the following lemma.

Lemma 3 (Eventual agreement)**.** *There exists a point in time after GST, after which every correct oracle has initialized the same epoch e and does not abort it unless some correct oracle times out or an indicates a leader change for epoch e.*

Lemma 4 (Eventual Succession)**.** *Consider a period of network synchrony after GST. If more than f correct oracles are in epoch e with leader p_ℓ and either time out on p_ℓ or receive a changeleader event, then all correct oracles will advance to epoch $e + 1$ after at most $2\Delta_{resend} + 2\Delta + 4\epsilon$.*

Proof. Notice that when a correct oracle times out on p_ℓ or receives a *changeleader* event during epoch e, it sets $ne = e + 1$. Thus, the proof of Lemma 2 applies for this situation as well because more than f correct oracles send NEWEPOCH messages with $e + 1$. Hence, all correct oracles have initialized epoch $e + 1$ after at most $2\Delta_{\text{resend}} + 2\Delta + 4\epsilon$. ■

Lemma 5 (Putsch resistance)**.** *A correct oracle does not abort an epoch e unless at least one correct oracle has timed out on the leader of epoch e or the report generation protocol has indicated to change the leader.*

Proof. Notice that $\overline{ne}$ in Lemma 1 is the maximal epoch value that any correct oracle has ever sent in a NEWEPOCH message. Assume that e is the highest epoch initialized by any correct oracle, hence we have $\overline{ne} = e$.

Towards a contradiction, suppose that some correct oracle has aborted epoch e. According to the algorithm, this means it has received NEWEPOCH messages for some epoch higher than e from more than $2f$ oracles, of which some were sent by correct oracles. But by Lemma 1(a) with $\overline{ne} = e$, no correct oracle has sent a NEWEPOCH message with an epoch higher than e under the assumptions of putsch resistance. The lemma follows. ■

The statements of Lemmas 3–5 imply all properties of a pacemaker protocol. Therefore, the following result summarizes our analysis.

Theorem 6. *Algorithm 1 implements a pacemaker protocol.*

8.2 Report generation

Recall that the report generation protocol (Section 5.3) is parameterized by an epoch number e and a leader oracle p_ℓ. The protocol receives a *startepoch*(e, ℓ) event (only at the leader; the parameters e and ℓ are redundant). It generates one or more *transmit*(O) events at all oracles, where O denotes an attested report to be transmitted; we say that the oracle *starts transmission* at this point and the report is taken up by the transmission protocol. Moreover, the report generation protocol at every oracle periodically emits a *progress* event and thereby *signals that it makes progress* to the pacemaker protocol. The report generation protocol may also produce at most one *changeleader* event, indicating that the report generation protocol has ended; this occurs after transmitting $r_{\max}$ attested reports.

startepoch(e, ℓ)**:** Instructs the oracle p_ℓ to start epoch e as leader.

progress**:** Indicates that the report generation protocol instance has made progress, in the sense that the leader p_ℓ has collected one round of observations from the oracles.

changeleader**:** Indicates that the report generation protocol terminates because it has performed the maximum foreseen number of observation collections and possible transmissions.

transmit(O)**:** The report generation protocol has generated an attested report O and starts transmission with O.

Furthermore, every oracle running the protocol has access to a constantly changing *observation value* v; we assume this is read into variable v immediately before the code accesses this value. We say that an oracle *observes value* v when this occurs. An oracle p_i may consult the most recent attested report C_O, which contains the oracle report C_R committed most recently by C. Notice that due to Ethereum's dissemination delays, not every oracle receives updates to C_O at the same time. In the following, we refer to this C_R as the *committed report that* p_i *knows.*

Internally, the leader of the report generation protocol uses two timers, T_{round} and T_{grace}. They are used to refine the above properties during synchronous periods. Time T_{round} runs continuously with duration Δ_{round} and controls the periodic start of a new report generation round. Timer T_{grace} is active during the grace period for Δ_{grace} and controls the extra waiting time for receiving observations into the current report from late oracles.

When an oracle p_i receives a report R containing observations from $2f + 1$ oracles, the oracle checks if R *should be reported* with respect to the committed report C_R and the current time, as specified by function *should-report*(R) in Algorithm 2. In the following, let $\Delta^+ = \Delta_{\text{grace}} + 6\Delta + 8\epsilon$.

Definition 2. *A* report generation *protocol satisfies these conditions:*

Transmission totality: *If at some point in time after GST a correct oracle starts transmission with an attested report* $O = (e, r, R, \dots)$ *and the report generation protocol instance is not aborted, then every correct oracle starts transmission with an attested report* $O' = (e', r', R', \dots)$ *such that* $(e, r) = (e, r)$ *within at most* $2\Delta + 2\epsilon$.

Observation integrity: *If a correct oracle starts transmission of an attested report with median* v*, then* v *is either the observation of a correct oracle or lies between the observations of two correct oracles.*

Responsiveness: *Consider a point in time after GST when the network has synchronized and every correct oracle knows some committed report C_R and suppose p_ℓ is correct. If every correct oracle observes a value that should be reported and the epoch is not aborted, then every correct oracle starts transmission within at most $\Delta_{round} + \Delta^+$.*

Progress: *Suppose p_ℓ is correct and that the report generation protocol is not aborted. Consider the point in time after GST, when all correct oracles have initialized the report generation protocol. Then, every correct oracle indicates that it makes progress or started a transmission once in the interval $[\rho\,(\Delta_{round} + \epsilon), \rho\,(\Delta_{round} + \epsilon) + \Delta^+]$ for $\rho = 1, \ldots, r_{max} - 1$. Furthermore, every correct oracle indicates to change the leader once in the interval $[r_{max}\,(\Delta_{round} + \epsilon), r_{max}\,(\Delta_{round} + \epsilon) + \Delta^+]$.*

Timely inclusion: *Consider a point in time after GST when the network has synchronized and suppose some correct oracle p_i observes a value v that should be reported. If the leader is correct and $\Delta + 2\epsilon \leq \Delta_{grace}$, then the attested report in the next transmission started by p_i contains v reported by p_i.*

Bounded leader tenure: *No correct oracle starts transmission in this report generation protocol more than r_{max} times.*

The remainder of this section is devoted to a proof that Algorithm 2–4 implements a report generation protocol.

Lemma 7 (Transmission totality)**.** *Consider a point in time after GST and suppose a correct oracle starts transmission in a round r. Then every correct oracle starts transmission in round r within at most $2\Delta + 2\epsilon$ or the report generation protocol instance is aborted.*

Proof. Suppose the correct oracle p_i has started transmission in round r and that the report generation instance is not aborted. Then it has received properly attested reports, as determined by *verify-attested-report*, from more than f distinct other oracles. Since the verification function does not depend on local state, every other correct oracle also recognizes such an attested report as properly attested.

Notice at least one of the oracles that sent a properly attested report to p_i is correct, hence, it has sent a FINAL-ECHO message containing this attested report O to all nodes. At most $\Delta + \epsilon$ later, every correct node therefore has received O or has already received some properly attested report earlier. This means that every correct node sends a FINAL-ECHO with O in response or has sent some attested report in such a message earlier. Thus, at the latest after $2\Delta + 2\epsilon$, every correct oracles has received $n - f > f$ FINAL-ECHO messages with possibly different, but always valid, attested reports and has started to transmit a report. ■

Lemma 8 (Observation integrity)**.** *The median observation value in an attested report as determined by verify-attested-report is either the observation of a correct oracle or lies between the observations of two correct oracles.*

Proof. Recall that an attested report R contains a (compressed) list of $2f + 1$ observations. Since R has been attested, there is at least one correct oracle p_j that has validated and signed R. During the report validation earlier, p_j has verified that every observation in R has been signed by a distinct oracle.

Since the report contains $2f + 1$ observations, strictly more than half of the values in R represent observations by correct oracles. Hence, the median value of an attested report is either an observation of a correct oracle or it is from a faulty oracle but R contains a smaller and a larger observation, both from correct oracles. ■

Lemma 9 (Responsiveness)**.** *Consider a point in time after GST when the network has synchronized and every correct oracle knows some committed report C_R and suppose p_ℓ is correct. If every correct oracle observes a value that should be reported and the epoch is not aborted, then every correct oracle starts transmission within at most $\Delta_{round} + \Delta_{grace} + 6\Delta + 8\epsilon$.*

Proof. Notice that under the conditions of the lemma, a fresh round is started by the correct leader at most after Δ_{round}. Every correct oracle sends its observation to p_ℓ, and p_ℓ combines these to a report R. According to Lemma 8, the median observation value in R is the observation of a correct oracle or lies between the observations of two correct oracles. Hence, testing if the median value of R should be reported ensures that when all correct oracles observe values to be reported, then the test succeeds.

For the rest of the argument, follow the algorithm as driven by the correct leader. Then the additional maximal delays after Δ_{round} until a correct oracle transmits the report are as follows:

1) $\epsilon + \Delta$ for p_ℓ to start the round and send the OBSERVE-REQ message until every correct oracle receives it;

2) $\epsilon + \Delta$ for every correct oracle to respond with a OBSERVE message to p_ℓ;

3) $\epsilon + \Delta_{\text{grace}}$ for the leader to process all OBSERVE-REQ messages and to wait during the grace period;

4) $\epsilon + \Delta$ for p_ℓ to finish the grace period and to send a REPORT-REQ message to all oracles until every correct oracle receives this;

5) $\epsilon + \Delta$ for every correct oracle to respond with a REPORT message to p_ℓ;

6) $\epsilon + \Delta$ for p_ℓ to produce a FINAL message and until every correct oracle receives it;

7) $\epsilon + \Delta$ for every correct oracle respond to the FINAL message with a FINAL-ECHO message and until every correct oracle receives $f + 1$ of these; and

8) ϵ for every correct node to process the FINAL-ECHO messages and start transmission.

■

The precondition of the previous lemma can be generalized in the sense that not all correct oracles may locally know the *same* report C_R. In particular, each one might know a potentially different committed report from C, but still such that the report value R in REPORT-REQ should be reported compared to the committed report. Then the conclusion of the lemma follows as well. Last but not least, recall from the implementation of *should-report*(R) that to check whether R should be reported, the time that has elapsed since the latest known committed report was produced and moment of checking counts as well.

Lemma 10 (Progress)**.** *Suppose p_ℓ is correct and that the report generation protocol is not aborted. Consider the point in time after GST, when all correct oracles have started the report generation protocol, and let $\Delta^+ = \Delta_{grace} + 6\Delta + 8\epsilon$. Then, every correct oracle has indicated that it makes progress or started a transmission once in the interval $[\rho\,(\Delta_{round} + \epsilon), \rho\,(\Delta_{round} + \epsilon) + \Delta^+]$ for $\rho = 1, \ldots, r_{max} - 1$. Furthermore, every correct oracle indicates to change the leader once in the interval $[r_{max}\,(\Delta_{round} + \epsilon), r_{max}\,(\Delta_{round} + \epsilon) + \Delta^+]$.*

Proof. The proof of Lemma 9 above shows that the duration of a round at each correct oracle is at most $\Delta^+ = \Delta_{\text{grace}} + 6\Delta + 8\epsilon$, measured from the time when the leader starts it. Since clocks are synchronized and messages are not delayed by more than Δ after GST, p_ℓ starts round number ρ with a delay of $\rho\,(\Delta_{\text{round}} + \epsilon)$ after starting the report generation protocol. According to the protocol and using similar reasoning as in the proof above, every correct oracle then indicates either progress or a leader change at most Δ^+ after the leader starts the round. ■

Lemma 11 (Timely inclusion)**.** *Consider a point in time when the network has synchronized and suppose some correct oracle p_i observes a value v that should be reported. If the leader is correct and $\Delta + \epsilon \leq \Delta_{grace}$, then the attested report in the next transmission started by any correct oracle contains v reported by p_i.*

Proof. Considering that this property is only guaranteed after GST, the inclusion follows directly from the implementation of the grace period: The leader waits at least for Δ_{grace} until it assembles the report R for a round. This allows correct oracle p_i to include its observation in R, which takes up to $\Delta + 2\epsilon$ for preparing and sending the OBSERVE message and for processing this by p_ℓ. ■

Lemma 12 (Bounded leader tenure)**.** *No correct oracle starts transmission in this report generation protocol more than r_{max} times.*

Proof. This property follows directly from the protocol, since every correct oracle p_i counts the number of rounds in a local integer variable r. In particular, p_i initializes r to 0 and subsequently verifies that when an OBSERVE-REQ message arrives and the leader announces a new round, any change to r only increments r. Furthermore, p_i halts the report generation when p_ℓ attempts to start a round with number greater than $r_{\max}$. ■

Observe that Lemmas 7–12 correspond to the definition of a report generation protocol. Therefore, the following result holds.

Theorem 13. *Algorithm 2–4 implements a report generation protocol.*

8.3 Transmission

There is one single instance of the transmission protocol that uses two events. The protocol is defined with respect to synchronous time. A *transmit*(O) event may be invoked on the transmission protocol to receive an attested report O for transmission. The algorithm performs various checks and if these succeed, it *sends* a transaction containing O on the Ethereum network to C. More formally, we consider two events.

***transmit*(O):** Receives an attested report O for transmission.

***send transaction* with O:** Sends a Ethereum transaction to C with O.

Every oracle p_i locally receives a sequence of reports for transmission, where each report is identified by a unique epoch-round tuple according to the implementation of report generation. When the properties below refer to an *order* among reports, this always corresponds to the order given by these epoch-round tuples.

For reports received for transmission, this order is the same as the order of the transmission events, according to the properties of the report generation protocol in combination with the pacemaker. However, *not* every correct oracle receives the same *sequence of reports* for transmission.

The transmission protocol internally buffers every report and sends a corresponding transaction only after a predetermined delay. The protocol also has access to the most recently committed report by C on the blockchain, which is denoted by C_O.

Definition 3. *A* transmission *protocol satisfies for each oracle p_i:*

Liveness: *When p_i receives a report O for transmission and either the median observation in O differs sufficiently from the observation in the most recently buffered report L or C_O is already at least as recent as L, then p_i buffers O.*

Furthermore, for every buffered report O, an Ethereum transaction is sent after a predetermined delay, unless C_O has meanwhile become at least as recent as O.

Safety: *No Ethereum transaction is sent with a report O unless O has been received for transmission.*

Theorem 14. *Algorithm 5 constitutes a transmission protocol.*

Proof. The liveness and the safety properties follow directly from the implementation. ■

Latency and cost. We now investigate the delay before the reports received for transmission are sent in transactions to C and the total cost incurred by the transactions executed by C on the blockchain. Here we deviate from the model of the adaptive adversary, and instead we assume the adversary chooses which oracles to compromise before the secret key x for the pseudo-random delay function becomes known to the oracles and the adversary.

Recall that Algorithm 5 operates in stages according to a schedule and buffers reports for multiples of a predetermined time Δ_{stage}. For convenience, we define $s_{1..j} = \sum_{i=1}^{j} s_i$.

If none of the f faulty oracles ever sends a transaction, we have

$$\begin{aligned}&\mathrm{P}\big[\text{transmission takes longer than } k \cdot \Delta_{\text{stage}}\big] = \\ &\mathrm{P}\big[\text{no successful transmission in stages } 1..k\big] = \begin{cases} \frac{f!(n-s_{1..k})!}{(f-s_{1..k})!n!} & \text{if } s_{1..k} \leq f \\ 0 & \text{otherwise} \end{cases},\end{aligned}$$

by uniformly sampling from the oracle set without replacement.

Let c be the cost of the first transaction sent to C on Ethereum and let c' be the cost of subsequent transactions that fail. In expectation, the cost of a given schedule is

$$\mathrm{E}\big[\text{transmission cost}\big] = (c - c') + \sum_{i=1}^{|S|} \mathrm{P}\big[\text{no successful transmission in stages } 1..i-1\big]\ s_i c'.$$

A simulation tool has been developed to illustrate the sending of transactions according to a schedule. The file `transmission_probability.py` contains a Python program for calculating latency and cost of different schedules.

For example, for $S = (2, 2, 7, 20), n = 31, f = 10$, the output is:

```
Have 31 replicas of which 10 are faulty

Reach stage 0 with p=1.0
Reach stage 1 with p=0.09677
Reach stage 2 with p=0.006674
Reach stage 3 with p=0.0

Expected cost is 486818 gas
```

References

[BCG20] Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. "Making Byzantine Consensus Live". In: *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*. Ed. by Hagit Attiya. Vol. 179. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 23:1–23:17. DOI: 10.4230/LIPIcs.DISC.2020.23. URL: https://doi.org/10.4230/LIPIcs.DISC.2020.23.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3. URL: https://doi.org/10.1007/978-3-642-15260-3.

[CL02] Miguel Castro and Barbara Liskov. "Practical byzantine fault tolerance and proactive recovery". In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461. DOI: 10.1145/571637.571640. URL: https://doi.org/10.1145/571637.571640.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323. DOI: 10.1145/42282.42283. URL: http://doi.acm.org/10.1145/42282.42283.

[EJN17] Steve Ellis, Ari Juels, and Sergey Nazarov. *ChainLink: A Decentralized Oracle Network*. https://chain.link/whitepaper. 2017.

[JBT20] Ari Juels, Lorenz Breidenbach, and Florian Tramèr. *Fair Sequencing Services: Enabling a Provably Fair DeFi Ecosystem*. https://blog.chain.link/chainlink-fair-sequencing-services-enabling-a-provably-fair-defi-ecosystem. 2020.

[NK20] Oded Naor and Idit Keidar. "Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR". In: *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*. Ed. by Hagit Attiya. Vol. 179. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 26:1–26:17. DOI: 10.4230/LIPIcs.DISC.2020.26. URL: https://doi.org/10.4230/LIPIcs.DISC.2020.26.