Chainlink Offchain Reporting Protocol 3.0

Lorenz Breidenbach* Christian Cachin† Alex Coventry Yan Ji Kostis Karantias
Philipp Schindler Chrysoula Stathakopoulou Alexandru Topliceanu

Chainlink Labs

18 May 2025

1 Introduction

The Chainlink Offchain Reporting Protocol (OCR) is a Byzantine fault tolerant distributed protocol run amongst a network of n *oracles* (or *nodes*). The protocol repeatedly gathers *observations* from participating oracles and aggregates them into consensus *outcomes*. From each outcome, multiple *reports* are derived and attested by a quorum of oracles (through their signatures). The attested reports are then transmitted by the oracles to an *oracle smart contract* C, running on a blockchain, which could be Ethereum or any other blockchain which supports smart contracts. The contract validates the reports and exposes their contents to consuming contracts.

The Offchain Reporting Protocol can execute many different kinds of functions through *reporting plugins*. A reporting plugin is a stateful object, implementing functions that define how individual oracles make *observations*, how the observations are aggregated into an *outcome*, how the outcome is converted into a *report*, whether the report should be *transmitted* to C, and more. For a given instance of the protocol, all oracles are assumed to run the same reporting plugin. Whenever we mention observations, aggregate outcomes, and reports in this document, they are meant in the generic sense, without taking into account their specific meaning in the context of a reporting-plugin implementation. The logic of each reporting plugin is executed offchain on the oracles participating in the protocol. The smart contract C is the onchain counterpart of the reporting plugin. Its logic is executed onchain.

By way of (imperfect) analogy, consider the programming model of MapReduce [DG04] with its separation between logic expressed in terms of map and reduce functions and a runtime executing these functions without having to be aware of what data is being mapped and reduced. Similarly, for the Offchain Reporting Protocol, logic is expressed following the structure of the reporting plugin interface and the protocol acts as a distributed, Byzantine fault tolerant runtime for that logic without being aware of what is being observed, aggregated, and transmitted.

Major protocol versions. The first version of the protocol, OCR1, was released in February 2021 and is documented in Version 1.2 of this document. OCR1 is specific to data feeds and its implementation only targets Ethereum-compatible blockchains. The next version of the protocol, OCR2, generalized the design of the protocol, so that it may collect and report not only numerical values (from an ordered set), but more general kinds of data.

This version, OCR3, strengthens the protocol with replicated state and ensures a total order among the reports it produces. Thanks to OCR3's optimistic responsiveness and focus on latency minimization, real-world production deployments of this protocol achieve end-to-end latencies in the low hundreds of milliseconds over the Internet. Moreover, OCR3's support for batched report generation improves report throughput versus OCR2 by three orders of magnitude (1000x).

In the rest of the document, for simplicity, we will use the acronym OCR to refer to Offchain Reporting Protocol version 3.0.

*Authors are listed in alphabetical order.

†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.

2 Design goals

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

Security & reliability: The protocol and its implementation 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. The design should be simple to implement in a real-world system.

Low transaction fees: Communication between the oracles and computation performed by the oracles happens offchain and is therefore (almost) free. In contrast, communication with C, and computation performed by it, require transactions on the blockchain, and are many orders of magnitude more expensive. For example, a single Ethereum transaction can easily cost tens or hundreds of dollars. We thus aim to minimize transaction fees, even if this results in a protocol with higher offchain computation and networking requirements.

Low latency: Oracles should provide data as fresh as possible. The design minimizes the delay between observations being made and transmission of the corresponding attested report, as well as the delay incurred by generation of new outcomes that causally depend on the previous outcome. One of the main ways in which we achieve this is by making the protocol responsive and reducing the number of network trips, since network latency is the dominant constituent of protocol latency in the WAN settings in which the protocol operates.

Flexibility: The protocol should act as a common foundation on top of which many different use cases can be built by configuring the protocol appropriately and "plugging in" use case specific logic through the reporting plugin and oracle contract. Implementers should not need to be familiar with the internals of the protocol but only with the reporting plugin interface.

3 Model

Oracles. The system consists of a set $\mathcal{P} = \{p_1, \ldots, p_n\}$ of n *nodes*, to which we refer as *oracles*, and the oracle contract C. 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 a *configuration smart contract*, which might be the same as contract C. The configuration contract also maintains the public keys of all participating oracles and makes these keys available to all. Each oracle p_i, through the reporting plugin, makes time-varying observations of a value, such as a price.

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 nonfaulty oracles are called *correct*.

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 with $n = 3f + 1$, since this gives optimal resilience.

In principle, the failure assumption also covers 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.

We also permit *benign* faults of nodes in the following sense: An otherwise 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.

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) must always satisfy the safety properties.

To support this refined model and thereby increase the resilience against crashes, some local state is always maintained in persistent storage. We will explicitly mention the variables for which this is the case.

Oracle smart contract and reports. The protocol repeatedly produces reports, with the goal of having them recorded on a blockchain by the smart contract C.

A configurable number of observations are aggregated into a sequenced outcome by a *Byzantine fault-tolerant (BFT) quorum* of oracles, i.e., a set of $\lceil \frac{n+f+1}{2} \rceil$ nodes. This ensures that outcomes are sequenced consistently across all correct oracles. Reports are derived from the aggregated outcomes as defined by the reporting plugin implementation. Each report is signed by $f + 1$ oracles to guarantee that it is signed by at least one correct oracle. The report is then submitted to C which, in turn, verifies the signatures, validates the report, records which oracles contributed, and stores the consolidated report on the blockchain. The contributing oracles receive a payout.

Cryptographic primitives. The protocol uses public-key digital signatures, pseudorandom functions (PRF), and cryptographic hash functions.

Digital signatures are implemented using standard elliptic-curve-based schemes; at least EdDSA and ECDSA are used, depending on context and the target blockchain. A protocol-internal digital signature is implemented by two operations, $\mathit{sign}_i(m)$ and $\mathit{verify}_j(m, \sigma)$. A call to sign_i must be executed by p_i, takes a bit string m as input, and returns a signature σ. The operation verify_j takes a string m and a potential signature σ as inputs and returns a Boolean. The implementation satisfies that $\mathit{verify}_i(\sigma, m)$ returns TRUE on any correct oracle if and only if p_i has executed $\mathit{sign}_i(m)$ and obtained σ before, except with negligible probability. The EdDSA scheme is typically used for this purpose. Attestations use a separate digital signature scheme with different keys; the respective operations are denoted by $\mathit{signAttest}_i(m)$ and $\mathit{verifyAttest}_j(m, \sigma)$. The attestation-signature scheme satisfies the same properties as the internal signatures and is typically implemented by ECDSA.

A cryptographic pseudorandom function (PRF) may be implemented by HMAC-SHA256 or by Keccak256 with prepending the key to the message. Formally, the PRF F_x maps strings of arbitrary length to strings of fixed length, and x is a secret key called the *seed*. The outputs of F_x cannot be efficiently distinguished from random bit strings by anyone who does not know the secret key.

SHA256 is used as a cryptographic collision-free hash function. Invoking it on a string s of arbitrary length is denoted by $H(s)$; this returns a fixed-length bit string. It is computationally infeasible to find collisions in H, i.e., no adversary can find s and $s' \neq s$ such that $H(s) = H(s')$ with non-negligible probability.

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

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, no oracle crashes, 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 6.6.

Network assumptions. The oracles may send point-to-point *messages* to each other over a network. All message transmissions are authenticated and encrypted, that is, each oracle can authenticate every other oracle based on the list of oracles as determined by the configuration contract on the blockchain.

Messages exchanged between correct oracles are always delivered in the same order in which they were sent, i.e., the protocol uses FIFO order message delivery. Protocols are described modularly through multiple algorithms that operate concurrently on one oracle. Arriving messages addressed to a particular algorithm are

also processed in the same sequence in which they are received. (The FIFO-order guarantee does not hold across algorithm instances.)

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 repeatedly try to send messages, all messages among correct nodes get through *eventually*, once any impeding network asynchrony passes.

The communication is implemented by one mutually authenticated TLS connection over the Internet for each pair of nodes. Messages sent among correct nodes are therefore delivered according to the reliable-delivery semantics of TCP. A node additionally maintains buffers for sending messages over the links and may store incoming messages in buffers until the protocol is ready to process them. These buffers have finite capacity and may therefore become full. In this case, the node drops messages that do not fit in a buffer. Hence, also correct but slow nodes may behave as if they have crashed when they drop messages (this behavior has also been called an omission fault). In the sense of the formal model, nodes that drop messages are treated as Byzantine.

Parts of the protocol contain explicit logic for resending messages. The other protocols transmit their point-to-point messages as described.

Notation. We give a semi-formal description of the protocol 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 communicates with other instances running on the same oracle through *events*, which are triggered by

invoke event *example-event*(*arg1*, *arg2*, $\cdots$) .

For each triggered event, a handler of the form

upon event *example-event*(*arg1*, *arg2*, $\cdots$) **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., in 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.

Sending a message to a protocol instance running on another oracle p_j is triggered by

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

Messages also trigger events on a destination node, denoted

upon receiving message [EXAMPLE-MESSAGE, *arg1*, *arg2*, $\cdots$] from p_j,

which let the protocol handle a message from oracle p_j.

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

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

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.

Finally, we make use of an event *scheduler*. It is accessed through the command

schedule event *example-event*(*arg1*, *arg2*, $\cdots$) after *time-period* .

which schedules the event *example-event* to be invoked once *time-period* of time has elapsed, starting from the moment of the invocation.

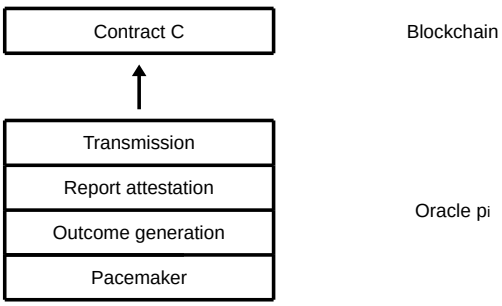


Figure 1. Architecture of the Offchain Reporting Protocol, Version 3.0.

4 Overview

The overall goal of the Chainlink Offchain Reporting Protocol is to periodically send reports to the contract C, which runs on a blockchain such as Ethereum. The reporting process is configurable through reporting plugins, of which each may provide its own semantics. The protocol is structured into four modular algorithms, called *pacemaker*, *(atomic) outcome generation*, *report attestation*, and *transmission*. All four protocols run continuously and concurrently to each other and are grouped as shown in Figure 1.

In a nutshell, the *(atomic) outcome generation protocol* is a leader-based protocol which executes in rounds. In each round, it attempts to aggregate observations into outcomes, based on a query and the previous outcome, such that each outcome is assigned a unique sequence number and the sequence of outcomes is contiguous. Every oracle computes this sequence locally, but a minority of the correct oracles may skip one or more outcomes in particular situations. Taking into account these potential omissions, the protocol ensures an "atomic" or total order among the outcomes that it generates at the correct oracles. In other words, it provides a service that comes close to total-order broadcast (which is often called consensus in the context of blockchains), but does not fully meet all guarantees of the latter.

The *pacemaker* ensures that the outcome generation protocol makes progress, even in case of a faulty leader or a network partition. It achieves this by rotating the outcome generation protocol leader through a succession of epochs such that all oracles are guaranteed to eventually be in the same epoch long enough to make progress.

The *report attestation protocol* converts aggregate outcomes into reports, according to the reporting plugin implementation, gathers signatures (an attestation) on the reports and hands the attested reports to the *transmission protocol*, which in turn sends the attested reports to C.

Pacemaker. The pacemaker protocol ensures the progress of the outcome generation protocol. The outcome generation protocol is structured into *epochs* and, in each epoch, a designated oracle acts as *leader* to drive the aggregation of observations into sequenced outcomes, similar to related protocols for consensus like PBFT [CL02] and HotStuff [Yin+19]. All oracles operate as *followers*.

The pacemaker protocol runs continuously and periodically initiates a new epoch within the outcome generation protocol. The pacemaker protocol instructs outcome generation to start a new epoch with a *newEpochStart* event. This occurs in two cases.

The first case is when the epoch has run for a ρ rounds with the same epoch leader. To balance load and trust, the leader should then change, and the outcome generation protocol emits a *newEpochReq* event, to which the pacemaker responds.

The second case occurs when the pacemaker protocol does not observe sufficient progress by the outcome generation protocol. To monitor this, the pacemaker receives *progress* events from outcome generation. Every oracle runs a timer T_{progress} with timeout duration Δ_{progress} that serves to watch the epoch leader's performance; every *progress* event resets that timer. When T_{progress} expires, the oracle concludes that the current leader is not performing correctly. The oracle then moves to initiate a new epoch with a different leader.

The outcome generation protocol also invokes *newEpochReq* events when the leader stays silent at the beginning of a new epoch for more than Δ_{initial} time.

In this way, the pacemaker protocol observes the progress of the current epoch within the outcome generation protocol only through *progress* events. Thus, Δ_{progress} and Δ_{initial} need not depend on the worst-case

transaction confirmation time on the blockchain.

Atomic outcome generation. The outcome generation protocol operates in epochs, and each epoch is denoted by a unique identifier e and associated to a leader node p_ℓ. Within each epoch, the protocol proceeds in rounds. In each round, the protocol gathers *observations* and, if conditions for going forward are met, generates an aggregate *outcome*, which it commits to a monotonically increasing *sequence number*, denoted by *sn*, using a PBFT message pattern [CL02]. Because of this sequence number, the protocol provides a notion of consensus or, more precisely, almost the guarantees of total-order broadcast.

The pace of round progress is controlled by the leader through a timeout Δ_{round}, triggered by the leader. It controls the minimum period of time the leader waits to start a successive round. The latency required to complete one round of the aggregation protocol during periods of synchrony and the value of Δ_{round} plus a safety margin must both be lower than Δ_{progress}.

Once a sufficient number of observations are aggregated into an outcome and the outcome has been committed to a sequence number, the outcome is handed over to the report attestation protocol. The outcome generation protocol ensures that no two distinct correct oracles commit different outcomes to the same sequence number within and across epochs. Every outcome is committed by at least $\lceil \frac{n+f+1}{2} \rceil - f \geq f + 1$ correct oracles, but not necessarily by all of them. Therefore, for each committed outcome, there exists at least one correct oracle that invokes the report attestation protocol with the same outcomes in the same order.

In principle, the contract C may verify that a report is valid. For preventing, however, unnecessary transmissions for invalid reports, the outcome may encode information on whether it converts into valid reports. The validity conditions depend on the implementation of the reporting plugin.

Report attestation. The report attestation protocol runs continuously and is responsible for gathering signatures on the reports that are derived. When a report is signed by $f + 1$ oracles, it is called *attested*. The report attestation protocol passes the attested reports to the transmission protocol which, in turn, transmits them to C. The contract C may verify the validity of a report by verifying the report attestation. An attestation of $f + 1$ signatures ensures that at least one correct oracle invoked the report attestation protocol with the outcome from which the report is derived and, therefore, the report satisfies the validity conditions specified in the plugin implementation as well as the safety properties guaranteed by the outcome generation protocol.

Transmission. The transmission protocol encapsulates the steps performed locally by each oracle for sending attested reports to C. Unlike the above algorithms, the transmission protocol does not involve any communication among the oracles. The transmission protocol delays each oracle by a pseudorandomly chosen waiting time to ensure a staged sending process. This aims at 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, even if only to discard them due to being outdated or duplicated.

5 Reporting plugin

A reporting plugin makes the Offchain Reporting Protocol configurable for multiple different purposes. The plugin abstracts the data that is reported by the offchain network and contains all functions for processing and filtering them. An example is the one for reporting numerical values, which is briefly described at the end of this section and in more detail in Section 7.

The functions are summarized in Algorithm 1 and consist of one function executed only by the outcome generation protocol leader, two processing functions executed by all oracles as followers in the outcome generation protocol, one processing function executed by all oracles in the report attestation protocol, one validation function for observations, two validation functions for reports, and a quorum size configuration function.

The $\mathit{query}(O_{\text{prev}}, sn)$ function is only run by the leader of the outcome generation protocol and determines a query Q, which is then sent to the followers, optionally based on the previous report O_{prev}. It may contain specific information on what data the followers should observe in the current round.

For gathering the actual data of the observation, the oracles invoke $\mathit{observation}(O_{\text{prev}}, sn, Q)$. It simply returns an opaque value v. It may be thought of as a simple number or as a complex data structure. A number of $\mathit{observation\text{-}quorum}(O_{\text{prev}}, sn, Q)$ signed observation values are needed to generate an outcome.

Algorithm 1 Interface of a reporting plugin

state
// A reporting plugin gets to define its own state that the various functions can access and modify.

function *query*(O_{prev}, *sn*)
// Executed only by the epoch leader.
// Provides a useful coordination mechanism in case there are many possible things that could be "observed."
// Produces a query that is sent to the followers.

function *observation*(O_{prev}, *sn*, Q)
// Produces an observation for sequence number *sn*.
// The observation may depend on the query Q and/or on the previous outcome O_{prev}.
// Returns an observation.

function *valid-observation*(O_{prev}, *sn*, Q, v)
// Validity conditions may depend on the query Q and/or on the previous outcome O_{prev}.
// Returns TRUE if the observed value v satisfies specific to the plugin validity conditions, FALSE otherwise.

function *observation-quorum*(O_{prev}, *sn*, Q)
// Returns the number of observations needed to build a report.
// This number may depend on the query Q and/or on the previous outcome O_{prev}.

function *outcome*(O_{prev}, *sn*, Q, B)
// Given the query Q and the previous outcome O_{prev}, aggregates the vector of observations B for sequence
// number *sn* according to an aggregation policy. Must be deterministic.
// Returns the aggregated outcome.

function *reports*(*sn*, O)
// Used by the report attestation protocol to transform the outcome O to a variable-length vector of reports.
// Returns a vector of reports.

function *should-accept-attested-report*(a)
// Used by the transmission protocol to decide whether an attested report should be accepted.
// This is invoked after a report has been attested.
// Returns a Boolean.

function *should-transmit-accepted-report*(a)
// Used by the transmission protocol to decide whether an attested report should be sent to C.
// This is invoked just before transmission potentially takes place.
// Returns a Boolean.

The *valid-observation*(O_{prev}, *sn*, Q, v) function determines whether the observation value v is valid. The leader uses this function to filter out invalid observations such that it sends a number of *observation-quorum*(O_{prev}, *sn*, Q) valid observations to the followers, if available. Followers use the function to verify the validity of the observations that they receive from the leader.

Each follower uses the deterministic function *outcome*(O_{prev}, *sn*, Q, B), which is specific to a query Q and may depend on the previous outcome O_{prev}, to aggregate a list B of values observed by different oracles to an outcome O.

The *reports*(*sn*, O) function is used by the report attestation protocol to transform the previously computed outcome O into a vector (variable-length array) of reports, such that the report attestation protocol collects signatures on all reports.

Finally, two filtering functions *should-accept-attested-report*(a) and *should-transmit-accepted-report*(a) are used by the transmission protocol for reducing the number of reports that are unnecessarily transmitted to contract C. Their parameter is an attested report a. For example, they can ensure that a report is not transmitted when C has already obtained another report that was produced more recently.

For reasoning about the progress of the configurable reporting protocol, we assume that every function of a reporting plugin returns within Δ_{process} time units.

Notice that all functions receive a sequence number *sn*, either explicitly or implicitly as part of an attested report (a). The details of how these functions are invoked are explained in Sections 6.3 and 6.5.

Reporting plugin for numerical values. The reporting plugin for numerical values produces reports that collect numerical values representing different observations of the same offchain signal, such as a price feed. We describe it in this document to provide a concrete example of a reporting plugin and also for continuity with the first version of the Offchain Reporting Protocol which could only produce reports about numerical values. This reporting plugin produces reports that represent the *median* value of more than $2f$ observations. More information about this plugin appears in Section 7.

6 Protocol

6.1 Contract

Reports generated by the OCR protocol are transmitted to C for onchain processing. This makes the contract the onchain counterpart to the offchain reporting plugin. Just like for the reporting plugin, the contract's logic depends on the purpose of the system being implemented.

The contract C also manages the configuration of the OCR protocol, that includes various configuration parameters and the set of participating oracles. This is required for the contract to check the authenticity of reports by cryptographically verifying report attestations. The contract informs the oracles of updated configurations by emitting events.

Reports contain the sequence number *sn* assigned by the atomic outcome generation protocol. More precisely, the plugin function $\mathit{reports}(\mathit{sn}, O)$ is used to derive a report vector from an outcome O, where the outcome generation protocol has committed *sn* to O. Every report is therefore associated with a tuple $(\mathit{sn}, \mathit{pos})$, where *pos* denotes the position of the report in the vector of reports.

This enables the report processing logic in the contract to inspect the sequence number *sn* of a report. A common pattern is for the contract to maintain a high-water mark of the greatest sequence number for which it has processed a report and ignore any reports with lesser sequence numbers as *stale*.

6.2 Pacemaker

The pacemaker protocol in Algorithm 2 on p. 10 governs progression through epochs numbered in succession $1, 2, 3, \ldots$ and the choice of the epoch leader. For each epoch, the outcome generation protocol in Algorithms 3–6 produces up to ρ outcomes, at most one per round. If the outcome generation protocol does not produce an outcome after Δ_{progress} units of time from the beginning of the epoch or the previous outcome, the oracle initiates a switch to the next epoch, and the corresponding next leader.

Description. Algorithm 2 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 [BCG20; NK24].

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 NEW-EPOCH-WISH message. These variables are maintained on persistent storage.

The node broadcasts a NEW-EPOCH-WISH message containing *ne* every Δ_{resend} seconds. This increases the probability that relevant NEW-EPOCH-WISH 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.

The NEW-EPOCH-WISH message in Algorithm 2 basically 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 committed an outcome by timeout Δ_{progress}.

The oracle stores the highest epochs received from all other oracles through NEW-EPOCH-WISH messages in an array *newEpochWishes*. If an oracle receives more than f messages of the form $[\text{NEW-EPOCH-WISH}, e']$, each one containing some $e' > e$, it infers that at least one correct node wishes to progress to some epoch $\bar{e}$ higher than e. The node chooses $\bar{e}$ as the $(f+1)$-highest entry of *newEpochWishes* and sends out its own $[\text{NEW-EPOCH-WISH}, \bar{e}]$ 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 $[\text{NEW-EPOCH-WISH}, e']$ message with $e' \geq \bar{e}$.

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 close to $2/3$ of the nodes positively respond with NEW-EPOCH-WISH 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 NEW-EPOCH-WISH messages. If a node observes $2f$ nodes wish to change to an epoch greater than e, the node switches to epoch $\bar{e}$, where $\bar{e}$ is the $(2f+1)$-highest entry of *newEpochWishes*. It then triggers a corresponding *newEpochStart*$(\bar{e}, \ell)$, which will be picked up by the outcome generation protocol.

Because more than $2f$ nodes 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 NEW-EPOCH-WISH messages as well, since a correct node will send this message to all others. Hence, all correct nodes will eventually transmit a NEW-EPOCH-WISH message containing epoch at least $\bar{e}$, according to the NEW-EPOCH-WISH amplification rule, and move to a new epoch $\bar{e}$ outcome generation protocol.

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 recovery from crashes solely in the context of Algorithm 2. When a node resumes after a crash, it restarts all running algorithms and restores certain variables from persistent storage.

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 NEW-EPOCH-WISH 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 NEW-EPOCH-WISH 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 pacemaker protocol will resume operations successfully once $n - f$ oracles are operating and more than f among them send a NEW-EPOCH-WISH message with an epoch value of at least some e after recovery. This ensures that, eventually, more than f oracles send their own NEW-EPOCH-WISH 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 informally why the pacemaker protocol is able to resume after the crash and subsequent recovery of any number of correct nodes. A more formal argument appears in Section 8.

Observe first that the protocol ensures that for every correct oracle, the variable *ne* increases monotonically and, likewise, no entry in *newEpochWishes* ever decreases. This follows directly from the protocol.

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 *newEpochWishes* 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* the outcome generation protocol operates with a correct leader that has started the same epoch for *all* correct oracles, outcomes (and *progress* events) are produced faster than Δ_{progress}. Hence, no correct oracle times out

Algorithm 2 Pacemaker protocol structured into epochs (executed by every oracle p_i).

state
$(e, \ell) \leftarrow (1, \mathit{leader}(1))$: current epoch and leader
$\mathit{ne} \leftarrow e$: highest epoch which p_i has initialized, i.e., highest epoch for which it has sent a NEW-EPOCH-WISH message
$\mathit{newEpochWishes} \leftarrow [0]^n$: highest epoch received from p_j in a NEW-EPOCH-WISH 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 NEW-EPOCH-WISH messages

upon initialization **do**
invoke event $\mathit{newEpochStart}(e, \ell)$ // see outcome generation protocol in Alg. 3
start timer T_{progress}

upon event *progress* **do** // the current leader is progressing with the outcome generation protocol (see Alg. 6)
restart timer T_{progress}

upon event *timeout* from T_{resend} **do** // resend NEW-EPOCH-WISH message every Δ_{resend} seconds
$\mathit{sendNewEpochWish}()$

upon event *timeout* from T_{progress} **or event** *newEpochReq* **do** // abort epoch because leader is too slow or tenure is over
stop timer T_{progress}
$\mathit{ne} \leftarrow \max\{e + 1, \mathit{ne}\}$
$\mathit{sendNewEpochWish}()$

upon receiving a message $[\text{NEW-EPOCH-WISH}, e']$ from p_j **do**
$\mathit{newEpochWishes}[j] \leftarrow \max(e', \mathit{newEpochWishes}[j])$

upon $\big|\{p_j \in \mathcal{P} \mid \mathit{newEpochWishes}[j] > \mathit{ne}\}\big| \geq f + 1$ **do** // NEW-EPOCH-WISH amplification rule
$\bar{e} \leftarrow \max\Big\{e' \Big| \big|\{p_j \in \mathcal{P} \mid \mathit{newEpochWishes}[j] \geq e'\}\big| \geq f + 1\Big\}$
$\mathit{ne} \leftarrow \max(\mathit{ne}, \bar{e})$
$\mathit{sendNewEpochWish}()$

upon $\big|\{p_j \in \mathcal{P} \mid \mathit{newEpochWishes}[j] > e\}\big| \geq 2f + 1$ **do** // agreement rule
$\bar{e} \leftarrow \max\Big\{e' \Big| \big|\{p_j \in \mathcal{P} \mid \mathit{newEpochWishes}[j] \geq e'\}\big| \geq 2f + 1\Big\}$
$(e, \ell) \leftarrow \big(\bar{e}, \mathit{leader}(\bar{e})\big)$
$\mathit{ne} \leftarrow \max\{\mathit{ne}, e\}$
restart timer T_{progress}
invoke event $\mathit{newEpochStart}(e, \ell)$ // see outcome generation protocol in Alg.3

function $\mathit{sendNewEpochWish}()$
send message $[\text{NEW-EPOCH-WISH}, \mathit{ne}]$ to all $p_j \in \mathcal{P}$
restart timer T_{resend}

on T_{progress} and initializes a further epoch like this. Moreover, oracles receive the first message of the epoch from a correct leader in less time than Δ_{initial} from the beginning of the epoch. Hence, no correct oracle times out on T_{initial}.

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 NEW-EPOCH-WISH messages, every correct node will soon receive $n - f$ NEW-EPOCH-WISH messages and determine an epoch number $\bar{e}$ in the NEW-EPOCH-WISH 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 invoked a *newEpochStart* event.

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 NEW-EPOCH-WISH 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 NEW-EPOCH-WISH messages and send a NEW-EPOCH-WISH message with parameter e_s or higher as well.

This implies that every correct oracle eventually stores $n - f > 2f$ entries in *newEpochWishes* that are at least e_s. Hence, every correct oracle has either already invoked a *newEpochStart* event for epoch e_s or will progress to epoch e_s and invoke a *newEpochStart* event for epoch e_s.

It remains to show that no correct oracle has yet progressed to some epoch $e' > e_s$. 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 *newEpochWishes* containing e' or a higher value. Hence, accounting for f values reported by faulty oracles, more than f correct oracles would have sent NEW-EPOCH-WISH 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 NEW-EPOCH-WISH 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 $\mathit{leader} : \mathbb{N} \to \{1, \ldots, n\}$ 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.

A trivial implementation is to set

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

The ordering of the oracles is determined by the list in the configuration contract. 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.

We implement the *leader* function using a pseudorandom function (PRF) 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. We use the PRF to calculate a random permutation π of $\{p_1, \ldots, p_n\}$ that applies to a span of n epochs. The leader of each epoch $i \in \{1, \ldots, n\}$ in the span is $\pi[i]$.

6.3 Atomic outcome generation

The outcome generation protocol proceeds in epochs, where each epoch consists of multiple rounds. We first describe the protocol proceeds through the rounds within an epoch and then how it switches to another epoch.

6.3.1 Rounds of an epoch

All oracles start a new epoch e upon receiving a $\mathit{newEpochStart}(e, \ell)$ event, where ℓ identifies a dedicated oracle which acts as epoch leader. All oracles, including the epoch leader, act as followers. Each epoch runs until the pacemaker determines that the subsequent epoch should be started or until it has executed ρ rounds.

Algorithm 3 Atomic outcome generation protocol, executed by oracle p_i (part 1)

state

$(e, \ell) \leftarrow (1, \mathit{leader}(1))$: current epoch and leader
sn $\leftarrow 0$: the sequence number for the current round
query $\leftarrow \perp$: the query for the current round
prevOutcome $\leftarrow \perp$: the outcome committed to sequence number *sn* $- 1$
outcome $\leftarrow \perp$: the outcome for the current round
firstSnOfEpoch $\leftarrow 0$: connects sequence numbers to the round numbers of the current epoch
prepareMsgs $\leftarrow [\perp]^n$: a vector of received PREPARE messages for the current round per oracle ID
commitMsgs $\leftarrow [\perp]^n$: a vector of received COMMIT messages for the current round per oracle ID
certOutcome $\leftarrow \perp$: the highest certified outcome (committed or else prepared)
certOutcomeQC $\leftarrow [\perp]^n$: the quorum of COMMIT or else PREPARE messages for the highest certified outcome
(*certEpoch*, *certSn*, *certTag*) $\leftarrow (0, 0, \perp)$: the highest certified timestamp, i.e., the epoch, sequence number, and message tag of *certOutcome*, where *certTag* $\in \{$COMMIT, PREPARE$\}$
phase $\leftarrow \perp$: denotes the phase in Φ as a follower within a round
timer T_{initial} with timeout duration Δ_{initial}, initially stopped

// state only updated by the epoch leader
timer T_{round} with timeout duration Δ_{round}, initially stopped
timer T_{grace} with timeout duration Δ_{grace}, initially stopped
waited $\leftarrow$ TRUE: denotes whether the next round may start because T_{round} has expired
observations $\leftarrow [\perp]^n$: vector of observations received for the current round
newEpochReqMsgs $\leftarrow [\text{FALSE}]^n$: a vector indicating if a EPOCH-START-REQ message is received in the current epoch per oracle ID
EPOCH-START-REQ messages
highCertOutcome $\leftarrow \perp$: highest prepared or committed outcome from EPOCH-START-REQ messages
highQC $\leftarrow [\perp]^n$: highest prepare- or commit-quorum certificate from EPOCH-START-REQ messages
(*highCertEpoch*, *highCertSn*, *highCertTag*) $\leftarrow (0, 0, \perp)$: the highest certified timestamp at the leader, i.e., the epoch, sequence number, and message tag of *highCertOutcome*, where *certTag* $\in \{$COMMIT, PREPARE$\}$
highQCProof $\leftarrow [\perp]^n$: highest prepare- or commit-quorum certificate proof for the current epoch
leaderPhase $\leftarrow \perp$: denotes the phase in Λ as a leader within a round

upon event *newEpochStart* (e', ℓ') **do**
$(e, \ell) \leftarrow (e', \ell')$
start timer $T_{initial}$
$\sigma \leftarrow \mathit{sign}_i(\text{EPOCH-START-REQ} \| e \| \mathit{certEpoch} \| \mathit{certSn} \| \mathit{certTag})$
send message [EPOCH-START-REQ, e, (*certEpoch*, *certSn*, *certTag*), *certOutcome*, *certOutcomeQC*, σ] to p_ℓ
phase $\leftarrow$ NEW-EPOCH
if $i = \ell$ **then** // executed only by epoch leader p_ℓ
newEpochReqMsgs $\leftarrow [\text{FALSE}]^n$
highCertOutcome $\leftarrow \perp$
highQC $\leftarrow [\perp]^n$
highQCProof $\leftarrow [\perp]^n$
(*highCertEpoch*, *highCertSn*, *highCertTag*) $\leftarrow (0, 0, \perp)$
leaderPhase $\leftarrow$ NEW-EPOCH

Algorithm 4 Atomic outcome generation protocol, executed by oracle p_i (part 2)

upon receiving message $[\text{EPOCH-START-REQ}, e', (\mathit{certEpoch}', \mathit{certSn}', \mathit{certTag}'), \mathit{outcome}', QC, \sigma]$ from p_j **s.t.**
$i = \ell \wedge e' = e \wedge \neg \mathit{newEpochReqMsgs}[j] \wedge \mathit{leaderPhase} = \text{NEW-EPOCH}$ **do** // only epoch leader p_ℓ
  $\mathit{newEpochReqMsgs}[j] \leftarrow \text{TRUE}$
  $\mathit{highCertTimestamp} \leftarrow (\mathit{highCertEpoch}, \mathit{highCertSn}, \mathit{highCertTag})$
  if $\mathit{verify}_j\big(\text{EPOCH-START-REQ} \| e' \| \mathit{certEpoch}' \| \mathit{certSn}' \| \mathit{certTag}', \sigma\big)$
      $\wedge\ \mathit{validQC}(\mathit{certEpoch}', \mathit{certSn}', \mathit{certTag}', \mathit{outcome}', QC)$ **then**
    if $\mathit{Less}(\mathit{highCertTimestamp}, (\mathit{certEpoch}', \mathit{certSn}', \mathit{certTag}'))$ **then**
      $\mathit{highCertTimestamp} \leftarrow (\mathit{certEpoch}', \mathit{certSn}', \mathit{certTag}')$
      $(\mathit{highCertEpoch}, \mathit{highCertSn}, \mathit{highCertTag}) \leftarrow \mathit{highCertTimestamp}$
      $(\mathit{highCertOutcome}, \mathit{highQC}) \leftarrow (\mathit{outcome}', QC)$
    $\mathit{highQCProof}[j] \leftarrow ((\mathit{certEpoch}', \mathit{certSn}', \mathit{certTag}'), \sigma)$
  if $\Big|\big\{p_j \in \mathcal{P} \big| \mathit{highQCProof}[j] \neq \perp\big\}\Big| = BQ(n, f)$ **then**
    send message $[\text{EPOCH-START}, e, \mathit{highCertTimestamp}, \mathit{highCertOutcome}, \mathit{highQC}, \mathit{highQCProof}]$ to all $p_j \in \mathcal{P}$
    if $\mathit{highCertTag} = \text{COMMIT}$ **then**
      $\mathit{leaderPhase} \leftarrow \text{SENT-EPOCHSTART}$
    else
      $\mathit{leaderPhase} \leftarrow \text{SENT-PROPOSAL}$

upon event *timeout* from T_{initial} **do**
  invoke event *newEpochReq* // see pacemaker protocol in Alg. 2

upon receiving message $[\text{EPOCH-START}, e', \mathit{highCertTimestamp}', \mathit{outcome}', \mathit{highQC}', \mathit{highQCProof}']$ from p_ℓ **s.t.**
$e' = e \wedge \mathit{phase} = \text{NEW-EPOCH}$ **do**
  if $\Big|\big\{p_j \in \mathcal{P} \big| \mathit{highQCProof}'[j] = (t', \sigma) \wedge t' = (\mathit{certEpoch}', \mathit{certSn}', \mathit{certTag}')$
      $\wedge\ \mathit{Less}(t', \mathit{highCertTimestamp}') \wedge \mathit{verify}_j(\mathit{certEpoch}' \| \mathit{certSn}' \| \mathit{certTag}', \sigma)\big\}\Big| \geq BQ(n, f)$
      $\wedge\ \mathit{validQC}(\mathit{certEpoch}', \mathit{certSn}', \mathit{certTag}', \mathit{outcome}', \mathit{highQC}')$ **then**
    $(\_, sn, tag) \leftarrow \mathit{highCertTimestamp}'$
    $\mathit{firstSnOfEpoch} \leftarrow sn + 1$
    $\mathit{outcome} \leftarrow \mathit{outcome}'$
    if $tag = \text{PREPARE}$ **then**
      $\sigma \leftarrow \mathit{sign}_i(\text{PREPARE} \| e \| sn \| H(\mathit{outcome}))$
      send message $[\text{PREPARE}, e, sn, \sigma]$ to all $p_j \in \mathcal{P}$
      $\mathit{phase} \leftarrow \text{SENT-PREPARE}$
    else // $tag = \text{COMMIT}$
      if $\mathit{Less}((\mathit{certEpoch}, \mathit{certSn}, \mathit{certTag}), (e, sn, \text{COMMIT}))$ **then**
        $\mathit{certOutcome} \leftarrow \mathit{outcome}$
        $\mathit{certOutcomeQC} \leftarrow \mathit{highQC}'$
        $(\mathit{certEpoch}, \mathit{certSn}, \mathit{certTag}) \leftarrow (e, sn, \text{COMMIT})$
        invoke event $\mathit{committedOutcome}(sn, (\mathit{outcome}, \mathit{commitQC}))$ // see report att. protocol in Alg. 7–8
      $\mathit{phase} \leftarrow \text{NEW-ROUND}$
    cancel timer T_{initial}

upon event *timeout* from T_{round} **s.t.** $i = \ell$ **do** // current round may be in any phase, executed only by epoch leader p_ℓ
  $\mathit{waited} \leftarrow \text{TRUE}$

upon $\mathit{waited} \wedge i = \ell$ // executed only by epoch leader p_ℓ
    $\wedge\ (\mathit{leaderPhase} = \text{SENT-EPOCHSTART} \vee \mathit{leaderPhase} = \text{COMMITTED})$ **do**
  $\mathit{query} \leftarrow \mathit{query}(\mathit{highCertOutcome}, sn + 1)$
  send message $[\text{ROUND-START}, e, sn + 1, \mathit{query}]$ to all $p_j \in \mathcal{P}$
  $\mathit{observations} \leftarrow [\perp]^n$
  start timer T_{round}
  $\mathit{leaderPhase} \leftarrow \text{SENT-ROUNDSTART}$
  $\mathit{waited} \leftarrow \text{FALSE}$

Algorithm 5 Atomic outcome generation protocol, executed by oracle p_i (part 3)

upon receiving message $[\text{ROUND-START}, e', sn', query']$ from p_ℓ **s.t.**
$e' = e \wedge sn' = sn + 1 \wedge phase = \text{NEW-ROUND}$ **do**
  if $sn' - firstSnOfEpoch > \rho$ **then** // p_ℓ has exhausted its maximal number of rounds
    invoke event *newEpochReq* // see pacemaker protocol in Alg. 2
    return
  $query \leftarrow query'$
  $v \leftarrow$ *observation*$(outcome, sn', query')$
  $\sigma \leftarrow sign_i(\text{OBSERVATION} \| e \| sn + 1 \| query' \| v)$
  send message $[\text{OBSERVATION}, e, sn + 1, v, \sigma]$ to p_ℓ
  $(prepareMsgs, commitMsgs) \leftarrow \big([\perp]^n, [\perp]^n\big)$
  $phase \leftarrow$ SENT-OBSERVATION

upon receiving message $[\text{OBSERVATION}, e', sn', v, \sigma]$ from p_j **s.t.** // executed only by epoch leader p_ℓ
$e' = e \wedge i = \ell \wedge sn' = sn + 1 \wedge observations[j] = \perp \wedge$ *valid-observation*$(prevOutcome, sn', query, v)$
$\wedge\ verify_j(\text{OBSERVATION} \| e' \| sn + 1 \| query \| v, \sigma) \wedge leaderPhase \in \{\text{SENT-ROUNDSTART}, \text{GRACE}\}$ **do**
  $observations[j] \leftarrow (v, \sigma)$

upon $|\{p_j \in \mathcal{P} | observations[j] \neq \perp\}| =$ *observation-quorum*$(prevOutcome, sn, query) \wedge i = \ell$
$\wedge\ leaderPhase = \text{SENT-ROUNDSTART}$ **do** // executed only by epoch leader p_ℓ
  start timer T_{grace} // grace period for slow oracles
  $leaderPhase \leftarrow$ GRACE

upon event *timeout* from T_{grace} **s.t.** $i = \ell \wedge leaderPhase = \text{GRACE}$ **do** // executed only by epoch leader p_ℓ
  $k \leftarrow 1$ // collect observations in B and signatures in Σ
  for $p_j \in \mathcal{P}$ **s.t.** $observations[j] \neq \perp \wedge observations[j] = (v, \sigma)$ **do**
    $B[k] \leftarrow (j, v)$
    $\Sigma[k] \leftarrow \sigma$
    $k \leftarrow k + 1$
  send message $[\text{PROPOSAL}, e, sn + 1, B, \Sigma]$ to all $p_j \in \mathcal{P}$
  $leaderPhase \leftarrow$ SENT-PROPOSAL

upon receiving message $[\text{PROPOSAL}, e', sn', B, \Sigma]$ from p_ℓ **s.t.**
$e' = e \wedge sn' = sn + 1 \wedge phase = \text{SENT-OBSERVATION}$ **do**
  $K \leftarrow$ *observation-quorum*$(prevOutcome, sn', query)$
  if $\bigwedge_{k=1}^{K} B[k] \neq \perp \wedge B[k] = (j, v) \wedge$ *valid-observation*$(prevOutcome, sn', query, v)$
  $\wedge\ verify_j\big(\text{OBSERVATION} \| e \| sn + 1 \| query \| v, \Sigma[k]\big)$ **then**
    $sn \leftarrow sn'$
    $prevOutcome \leftarrow outcome$
    $outcome \leftarrow$ *outcome*$(prevOutcome, sn, query, B)$ // *outcome* is deterministic
    $\sigma \leftarrow sign_i(\text{PREPARE} \| e \| sn \| H(outcome))$
    send message $[\text{PREPARE}, e, sn, \sigma]$ to all $p_j \in \mathcal{P}$
    $phase \leftarrow$ SENT-PREPARE

upon receiving message $[\text{PREPARE}, e', sn', \sigma]$ from p_j **s.t.** $e' = e \wedge sn' = sn \wedge prepareMsgs[j] = \perp$
$\wedge\ verify_j\big(\text{PREPARE} \| e \| sn \| H(outcome), \sigma\big) \wedge phase = \text{SENT-PREPARE}$ **do**
  $prepareMsgs[j] \leftarrow [\text{PREPARE}, e, sn, \sigma]$

Algorithm 6 Atomic outcome generation protocol, executed by oracle p_i (part 4)

upon $\left|\{p_j \in \mathcal{P} \mid \mathit{prepareMsgs}[j] \neq \bot\}\right| \geq BQ(n,f) \wedge \mathit{phase} = \text{SENT-PREPARE}$ **do**
  $\sigma \leftarrow \mathit{sign}_i(\text{COMMIT} \| e \| \mathit{sn} \| H(\mathit{outcome}))$
  send message $[\text{COMMIT}, e, \mathit{sn}, \sigma]$ to all $p_j \in \mathcal{P}$
  $\mathit{phase} \leftarrow \text{SENT-COMMIT}$
  if $\mathit{Less}((\mathit{certEpoch}, \mathit{certSn}, \mathit{certTag}), (e, \mathit{sn}, \text{PREPARE}))$ **then**
    $\mathit{certOutcome} \leftarrow \mathit{outcome}$
    $\mathit{certOutcomeQC} \leftarrow \mathit{prepareMsgs}$
    $(\mathit{certEpoch}, \mathit{certSn}, \mathit{certTag}) \leftarrow (e, \mathit{sn}, \text{PREPARE})$

upon receiving message $[\text{COMMIT}, e', \mathit{sn}', \sigma]$ from p_j **s.t.** $e' = e \wedge \mathit{sn}' = \mathit{sn} \wedge \mathit{phase} = \text{SENT-COMMIT}$
    $\wedge\ \mathit{commitMsgs}[j] = \bot \wedge \mathit{verify}_j\big(\text{COMMIT} \| e \| \mathit{sn} \| H(\mathit{outcome}), \sigma\big)$ **do**
  $\mathit{commitMsgs}[j] \leftarrow [\text{COMMIT}, e, \mathit{sn}, \sigma]$

upon $\left|\{p_j \in \mathcal{P} \mid \mathit{commitMsgs}[j] \neq \bot\}\right| \geq BQ(n,f) \wedge \mathit{phase} = \text{SENT-COMMIT}$ **do**
  $\mathit{phase} \leftarrow \text{NEW-ROUND}$
  if $p = \ell$ **then**
    $\mathit{leaderPhase} \leftarrow \text{COMMITTED}$
  if $\mathit{Less}((\mathit{certEpoch}, \mathit{certSn}, \mathit{certTag}), (e, \mathit{sn}, \text{COMMIT}))$ **then**
    $\mathit{certOutcome} \leftarrow \mathit{outcome}$
    $\mathit{certOutcomeQC} \leftarrow \mathit{commitMsgs}$
    $(\mathit{certEpoch}, \mathit{certSn}, \mathit{certTag}) \leftarrow (e, \mathit{sn}, \text{COMMIT})$
    invoke event $\mathit{committedOutcome}(\mathit{sn}, (\mathit{outcome}, \mathit{commitQC}))$ // see report attestation protocol in Alg. 7–8
  invoke event *progress* // indicates leader is performing correctly, see pacemaker protocol in Alg. 2

function $BQ(n, f)$
  return $\lceil \frac{n+f+1}{2} \rceil$

function $\mathit{validQC}(e, \mathit{sn}, \mathit{tag}, O, QC)$
  return $\exists?\, e'$ **s.t.** $e' = e \wedge \left|\left\{p_j \in \mathcal{P} \;\middle|\; QC[j] = [\mathit{tag}, e, \mathit{sn}, \sigma] \wedge \mathit{verify}_j\big(\mathit{tag} \| e \| \mathit{sn} \| H(O), \sigma\big)\right\}\right| \geq BQ(n,f)$

function $\mathit{Less}(t_1, t_2)$
  $(e_1, \mathit{sn}_1, \mathit{tag}_1) \leftarrow t_1$
  $(e_2, \mathit{sn}_2, \mathit{tag}_2) \leftarrow t_2$
  return $\mathit{sn}_1 < \mathit{sn}_2 \vee \mathit{sn}_1 = \mathit{sn}_2 \wedge \mathit{tag}_1 = \text{PREPARE} \wedge \mathit{tag}_2 = \text{COMMIT} \vee \mathit{sn}_1 = \mathit{sn}_2 \wedge \mathit{tag}_1 = \mathit{tag}_2 \wedge e_1 < e_2$

In each round all correct oracles should commit a single outcome O to a sequence number *sn*. Rounds are executed consecutively; a correct oracle does not enter a new round before committing an outcome to a sequence number for the previous round. Sequence numbers are contiguous and increase monotonically with the rounds of an epoch. Moreover, the sequence number is maintained across epochs. The first round of an epoch has a sequence number at least as high as the last round of some previous epoch(s).

Looking ahead, it will become clear that a minority of correct oracles, particularly those that have been slow in previous epochs, may have jumped to this epoch without committing an outcome in the last rounds of one (or more) previous epoch(s). Therefore, the sequence of committed outcomes at these oracles may contain a gap. However, it is ensured that for every sequence number, a majority among the correct oracles commits the outcome.

Every round is structured into phases. The epoch leader p_ℓ concurrently executes steps as the leader and as a follower. The follower progress is controlled by a variable *phase*, as for all oracles, whereas the progress of p_ℓ as a leader uses a variable *leaderPhase*.

The epoch's time duration is controlled by the pacemaker protocol. We say that an oracle finishes an epoch *gracefully* if it commits some outcome to a sequence number for all ρ rounds of the epoch, before the epoch expires. Otherwise, we say that the oracle finishes the epoch *ungracefully*.

At the beginning of a new epoch e, all oracles send to p_ℓ an EPOCH-START-REQ message as followers to inform the leader about their state in the previous epoch. The leader, upon receiving a BFT quorum, i.e., a

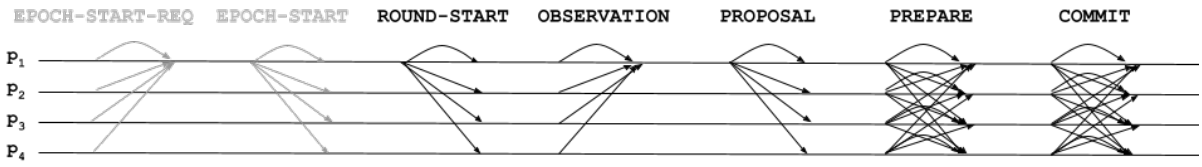


Figure 2. The first round of the outcome generation protocol in a fault-free execution with p_1 as epoch leader. Subsequent rounds start with the ROUND-START and previous messages (grey) are skipped.

set with cardinality $\lceil \frac{n+f+1}{2} \rceil$, of EPOCH-START-REQ messages sends a *new-epoch quorum certificate* (denoted *highQC*) in a EPOCH-START message. The new epoch quorum certificate determines a sequence number sn_{high} as well as an outcome O_{highCert} from some previous epoch that should be committed to sequence number sn_{high}. The leader starts the epoch with $sn = sn_{\text{high}} + 1$. This guarantees the consistency of the outcome generation protocol: if there exists some correct oracle which committed the outcome O_{highCert} to sequence number sn_{high} in some epoch $e' < e$, then no correct oracle commits a conflicting outcome to sn_{high} in epoch e' or higher. We defer the details of the EPOCH-START-REQ and EPOCH-START messages and the new-epoch certificate to Section 6.3.2.

Moreover, all oracles start a timer T_{initial} with duration Δ_{initial} at the beginning of a new epoch. Upon receiving a EPOCH-START message, an oracle cancels T_{initial}. The timer's duration Δ_{initial} should be shorter than Δ_{progress}, as this allows oracles to terminate an epoch fast in case the leader is not responsive right at the beginning of the epoch.

As the first step of each round, the leader sends a $[\text{ROUND-START}, e, sn', Q]$ message to all oracles, where e is the current epoch and $sn' = sn + 1$ is the sequence number for which the leader wants to generate an outcome in the current round. The variable *sn* is incremented only in a later phase of the round (after receiving the PROPOSAL message). The query Q is stored in *query* during the remainder of the round. The leader obtains Q by calling $\mathit{query}(O_{\text{highCert}}, sn')$, where O_{highCert} denotes the most recently committed outcome as determined by the leader, i.e., the outcome committed with sequence number $sn' - 1$. For controlling the pace of the protocol, the leader also starts a timer T_{round} that expires after Δ_{round}.

An oracle p_i accepts a message $[\text{ROUND-START}, e', sn', Q]$ if it is currently in epoch $e = e'$ and has committed an outcome for sequence number $sn = sn' - 1$. Upon accepting the ROUND-START message, p_i enters the new round. It evaluates $\mathit{observation}(O_{\text{prev}}, sn', Q)$, which returns an observation value v_i that may depend on the query Q and the previous outcome O_{prev}, i.e., the outcome committed with sequence number $sn - 1$. Then, p_i sends back to the leader the message $[\text{OBSERVATION}, e, sn', v_i, \sigma_i]$, where σ_i is p_i's signature on the OBSERVATION message and the query Q.

The leader waits for $\mathit{observation\text{-}quorum}(O_{\text{prev}}, sn', Q)$ valid OBSERVATION messages. An OBSERVATION message sent by oracle p_i for sequence number sn' is considered valid if it contains a valid signature σ_i and, moreover, the plugin function $\mathit{valid\text{-}observation}(O_{\text{prev}}, sn', Q, v_i)$ returns TRUE.

Having gathered $\mathit{observation\text{-}quorum}(O_{\text{prev}}, sn', Q)$ many valid OBSERVATION messages, the leader waits out a grace period of duration Δ_{grace} so that delayed observations may also be included in the report. When the grace period expires, the leader collects all observations in a vector B and all corresponding signatures in a vector Σ and sends them to the oracles in a $[\text{PROPOSAL}, e, sn, B, \Sigma]$ message.

When an oracle p_i receives the PROPOSAL message for the current epoch e and the sequence number *sn* of the round with a vector of $\mathit{observation\text{-}quorum}(O_{\text{prev}}, sn, Q)$ distinct and valid observations and a vector of valid corresponding signatures, it invokes the plugin function $\mathit{outcome}(O_{\text{prev}}, sn, Q, B)$. The latter returns an outcome O based on the vector of observations, on the query Q, and on the previous outcome O_{prev}, committed to sequence number $sn - 1$. The $\mathit{outcome}(\_\_)$ function is deterministic, so that all correct oracles obtain the same outcome O. Then p_i sends a message $[\text{PREPARE}, e, sn, \sigma_i]$ to all oracles, where σ_i is p_i's signature on $[\text{PREPARE}, e, sn, H(O)]$, where H denotes a cryptographic hash function. Note that no correct oracle will accept a PREPARE message before the corresponding PROPOSAL message and, hence, if the leader is correct, all correct oracles have the state needed to validate the signatures available locally.

When an oracle p_i has obtained a BFT quorum of PREPARE messages with valid signatures that match its local outcome O, it sends to all oracles a $[\text{COMMIT}, e, sn, \sigma_i]$ message, where σ_i is p_i's signature on

$[\text{COMMIT}, e, sn, H(O)]$, as before. When the first correct oracle does this, we say that the outcome is *prepared* in epoch e and for sequence number *sn*. When p_i receives a BFT quorum of matching COMMIT messages with valid signatures that also match its local state, the oracle *commits* its locally evaluated round outcome O to *sn* and O becomes *committed*. Then p_i passes *sn* and the *committed outcome*, i.e., the outcome O together with the quorum of signed COMMIT messages for O from epoch e, to the report attestation protocol. Moreover, p_i issues a *progress* event to the calling pacemaker protocol and completes the round.

We say that an outcome is *certified* when the outcome is either prepared or committed and accompanied by a BFT certificate of either PREPARE or COMMIT messages respectively. A certified outcome is additionally labeled with a *timestamp*, which we use to define an ordering relation among certified outcomes. A timestamp is a tuple $(certEpoch, certSn, certTag)$, where *certEpoch* and *certSn* are the matching epoch and sequence number of the PREPARE or COMMIT messages of the outcome certificate and $certTag \in \{\text{PREPARE}, \text{COMMIT}\}$ refers to the message tag of the messages in the certificate. Timestamps are first ordered by sequence number, then by message tag, where PREPARE $<$ COMMIT, and then by epoch. This order is implemented by $Less(\_, \_)$. Upon an outcome becoming prepared or committed, the oracle persists in storage the newly certified outcome O in a *certOutcome* variable, along with the corresponding certificate in *certOutcomeQC* variable, and its timestamp $(certEpoch, certSn, certTag)$, as long as the newly certified outcome has a timestamp greater than the timestamp of the previously persisted certified outcome.

After the epoch leader p_ℓ has completed the round, it becomes ready to initiate the next round. The leader starts a new round whenever the previous round has completed and Δ_{round} units of time have passed since starting the previous round, as controlled by a timer T_{round}. This means it waits for whichever event arrives later. The frequency of starting rounds therefore depends, to some extent, on the duration of executing rounds. By setting $\Delta_{\text{round}} = 0$, this enables the outcome generation protocol to commit outcomes as fast as the network permits.

The outcome generation protocol prevents that a leader runs for more than ρ rounds, where ρ is a global parameter. This is to avoid that a malicious leader drives the protocol forward as quickly as possible and causes a denial-of-service attack, for instance through oracles exhausting their computational or network capacity, or by making oracles hit some limits. In particular, when the leader attempts to start more than ρ rounds in the same epoch and a correct oracle receives a corresponding NEW-ROUND message as a follower, it does not start that round. Instead, the oracle signals this to the pacemaker protocol with a *newEpochReq* event and halts any further processing in the current epoch. It stays in phase NEW-ROUND, and the only way to continue the protocol is by responding to a *newEpochStart* event with a higher epoch number.

Alg. 3–6 describe how oracles aggregate observations to an outcome, commit to an outcome, and trigger the report attestation protocol with the committed outcome. The pseudocode mixes actions performed by the leader with those of the followers and orders them according to a failure-free run of the protocol. Every round of a follower is structured into (follower-)phases from a set

$$\Phi = \{\text{NEW-EPOCH}, \text{NEW-ROUND}, \text{SENT-OBSERVATION}, \text{SENT-PREPARE}, \text{SENT-COMMIT}\}$$

and obeys the phase transitions shown in Figure 3a.

The leader, concurrently to acting as a follower, also progresses according to the leader-phases from a set

$$\Lambda = \{\text{NEW-EPOCH}, \text{SENT-EPOCHSTART}, \text{SENT-ROUNDSTART}, \text{SENT-PROPOSAL}, \text{SENT-COMMIT}, \text{COMMITTED}\}.$$

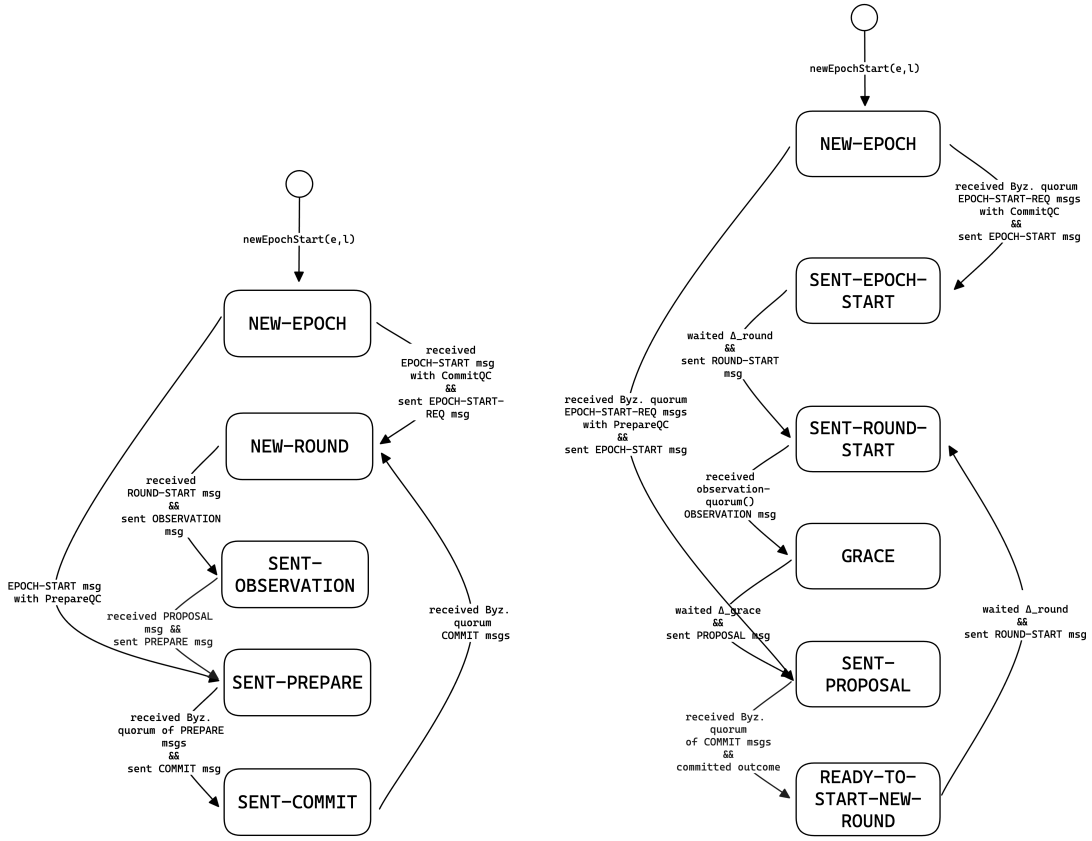
The leader operates according to the phase transitions of Figure 3b.

6.3.2 Start of a new epoch

A round may fail due to network delays or when the leader does not behave correctly. For example, the leader may send out conflicting PROPOSAL messages. Therefore, some oracle may not commit any outcome during the duration of a round. Such liveness violations are caught by the pacemaker protocol, because no *progress* events are emitted.

While the pacemaker protocol ensures that eventually all correct oracles are in the same epoch, it does not ensure consistency of the committed outcomes across epochs. To this end, at the beginning of a new epoch, all oracles send to the new epoch leader an EPOCH-START-REQ message with their highest certified outcome O_{cert} along with its corresponding certificate QC and its timestamp $(e_{\text{cert}}, sn_{\text{cert}}, tag_{\text{cert}})$. In detail, p_i sends a message

$$[\text{EPOCH-START-REQ}, e, (e_{\text{cert}}, sn_{\text{cert}}, tag_{\text{cert}}), O_{\text{cert}}, QC, \sigma]$$



(a) Transition of follower phases (in Φ). **(b)** Transition of leader phases (in Λ).

Figure 3. State diagrams of an oracle for the roles as follower and leader (for epochs where the oracle is leader) in the outcome generation protocol. All phases may additionally transition to the respective NEW-EPOCH phase via a *newEpochStart* event, this is not shown.

where

- e is the new epoch;
- $(e_{\text{cert}}, sn_{\text{cert}}, tag_{\text{cert}})$ is the timestamp of the oracle's locally persisted certified outcome;
- O_{cert} is the corresponding persisted certified outcome (committed or else prepared);
- *QC* the corresponding persisted BFT quorum certificate of COMMIT or else PREPARE messages; and
- σ is a signature of p_i on the string $e_{\text{cert}} \| sn_{\text{cert}} \| tag_{\text{cert}}$.

The leader, upon gathering $\lceil \frac{n+f+1}{2} \rceil$ valid EPOCH-START-REQ messages, determines a quorum certificate *highQC* for the new epoch to justify sn_{high} as the highest sequence number for which an outcome could have been committed and sends an EPOCH-START message to all oracles. In detail, sn_{high} is the sequence of the highest timestamp $(e_{\text{high}}, sn_{\text{high}}, tag_{\text{high}})$ among the valid EPOCH-START-REQ messages that the leader received; this means it is safe to start the first round of the new epoch with sequence number $sn_{\text{high}}+1$. The EPOCH-START message consists of

- e, the new epoch;
- the highest certified timestamp $(e_{\text{high}}, sn_{\text{high}}, tag_{\text{high}})$, which corresponds to sequence number sn_{high} of an outcome that could have been committed;
- the outcome O_{highCert} that corresponds to that timestamp;
- *highQC*, the highest commit else prepare quorum certificate among the $\lceil \frac{n+f+1}{2} \rceil$ EPOCH-START-REQ messages, which justifies O_{highCert} and the highest certified timestamp; and
- *highQCProof*, a BFT quorum of tuples (t_i, σ_i), where σ_i is the signature of p_i on the timestamp t_i of the EPOCH-START-REQ message from p_i, and more specifically, on $e_{\text{cert}} \| sn_{\text{cert}} \| tag_{\text{cert}}$, where $t_i = (e_{\text{cert}}, sn_{\text{cert}}, tag_{\text{cert}})$. This technique of proving that sn_{high} is indeed the highest sequence number that could have been committed may be viewed as an adaptation of Fast-HotStuff [JNF20] to our PBFT-style protocol.

When an oracle p_i receives a valid

$$[\text{EPOCH-START}, e, (e_{\text{high}}, sn_{\text{high}}, tag_{\text{high}}), O_{\text{highCert}}, highQC, highQCProof]$$

message for epoch e, timestamp $(e_{\text{high}}, sn_{\text{high}}, tag_{\text{high}})$, and *commit* quorum certificate (with $tag_{\text{high}} = \text{COMMIT}$), it first commits *highCertOutcome* to sn_{high}, if it has not done so, and passes $(sn_{\text{high}}, O_{\text{highCert}})$ to the report attestation protocol. Otherwise, the EPOCH-START message contains a *prepare* quorum certificate (with $tag_{\text{high}} = \text{PREPARE}$) for sn_{high} and O_{highCert}. In this case, p_i sends the message $[\text{PREPARE}, e, sn_{\text{high}}, \sigma_i]$ with σ_i being its signature on $\text{PREPARE} \| e \| sn_{\text{high}} \| O_{\text{highCert}}$ to all oracles. In either case, the oracle sets *firstSnOfEpoch* to $sn_{\text{high}} + 1$ and starts accepting ROUND-START messages for epoch e.

In the following let sn_{high} be this sequence number from the EPOCH-START-REQ message. Intuitively, if $sn_{\text{high}}+1$ is the lowest sequence number for which there exists no prepare quorum among $\lceil \frac{n+f+1}{2} \rceil$ EPOCH-START messages, then no correct oracle could have committed any outcome for $sn_{\text{high}}+1$, as there exist not enough oracles that could have sent a COMMIT message for $sn_{\text{high}}+1$. Therefore, $sn_{\text{high}}+1$ is free to assign a new outcome to. Moreover, no correct oracle could have prepared an outcome $O \neq O_{\text{highCert}}$ for sn_{high}, since there exists at least a prepare quorum certificate for O_{highCert} and, thus, no correct oracle could have committed $O \neq O_{\text{highCert}}$ for sn_{high} in a previous epoch. Therefore, it is safe to send a PREPARE message with outcome O_{highCert} for sn_{high} in the new epoch. Finally, upon receiving a valid commit certificate for sn_{high} and O_{highCert}, an oracle can directly commit, as any set of $\lceil \frac{n+f+1}{2} \rceil$ EPOCH-START-REQ messages will contain at least one prepare certificate for O_{highCert} and sn_{high}, and, therefore, no outcome different than O_{highCert} can be prepared, and hence committed, for sn_{high} in the new epoch.

Algorithm 7 Report attestation protocol, executed by oracle p_i (part 1)

state

availableCertOutcomes $\leftarrow \{\}$: maps a sequence number to a certified outcome
missingOutcomes $\leftarrow \emptyset$: a set of sequence numbers for which the outcome is missing
oraclesWithCertOutcomes $\leftarrow \{\}$: maps a sequence number to a randomized queue of oracle IDs
sentCertCommit $\leftarrow \{\}$: maps a sequence number to a set of oracle IDs to which CERTIFIED-COMMIT has been sent
reports $\leftarrow \{\}$: maps a sequence number to a vector of reports, non-$\perp$ as soon as REPORT-SIGS message is sent
reportSigMsgs $\leftarrow \{\}$: maps a sequence number to a vector of REPORT-SIGS messages per oracle ID
completeOutcomes $\leftarrow \emptyset$: set of sequence numbers corresponding to outcomes whose reports attestation completed

upon event *committedOutcome*(sn, CO) **do**
  if *availableOutcomes*$[sn] = \perp$ **then**
    availableCertOutcomes$[sn] \leftarrow CO$ // may trigger sending a REPORT-SIGS message

upon $\exists sn$ **s.t.** *availableCertOutcomes*$[sn] \neq \perp \wedge$ *reports*$[sn] = \perp$ **do**
  $(outcome, \_\_) \leftarrow CO$
  reports$[sn] \leftarrow$ *reports*$(sn, outcome)$ // *reports*$[sn]$ stores a vector of reports
  $nr \leftarrow |$*reports*$[sn]|$
  $\Sigma \leftarrow [\perp]^{nr}$
  for $k = 1, \ldots, nr$ **do**
    $\Sigma[k] \leftarrow$ *signAttest*$_i(sn,$ *reports*$[sn][k])$
  send message $[$REPORT-SIGS$, sn, \Sigma]$ to all $p_j \in \mathcal{P}$

upon receiving message $[$REPORT-SIGS$, sn, \Sigma]$ from p_j **s.t.** *reportSigMsgs*$[sn][j] = \perp$ **do**
  reportSigMsgs$[sn][j] \leftarrow [$REPORT-SIGS$, sn, \Sigma]$

upon $\left|\left\{p_j \in \mathcal{P} \middle| \textit{reportSigMsgs}[sn][j] = [\text{REPORT-SIGS}, sn, \Sigma] \neq \perp \right.\right.$
  $\left.\left. \wedge \bigwedge_{k=1}^{|\textit{reports}[sn]|} \Sigma[k] \neq \perp \wedge \textit{verifyAttest}_j\big((sn, \textit{reports}[sn][k]), \Sigma[k]\big) \right\}\right| \geq f + 1$
  $\wedge$ *reports*$[sn] \neq \perp \wedge sn \notin$ *completeOutcomes* **do**
  completeOutcomes $\leftarrow$ *completeOutcomes* $\cup \{sn\}$
  $nr \leftarrow |$*reports*$[sn]|$
  for $k = 1, \ldots, nr$ **do** // attest all reports
    $(h, j) \leftarrow (1, 1)$
    $(J, T) \leftarrow ([\perp]^{f+1}, [\perp]^{f+1})$
    while $h \leq f + 1$ **do** // collect the attestations for report
      while *reportSigMsgs*$[sn][j] = \perp$ **do** // at least $f + 1$ entries in *reportSigMsgs*$[sn]$ are non-$\perp$
        $j \leftarrow j + 1$
      $(\_\_, \_\_, \Sigma) \leftarrow$ *reportSigMsgs*$[sn][j]$
      $(J[h], T[h]) \leftarrow (j, \Sigma[k])$
      $h \leftarrow h + 1$
      $j \leftarrow j + 1$
    invoke event *attestedReport*$((sn, k,$ *reports*$[sn][k], J, T))$ // see transmission protocol in Alg. 9

6.4 Report attestation

The report attestation protocol is also run continuously by every oracle. It receives (sequence number, certified outcome) pairs from the outcome generation protocol, converts them into separate reports, gathers signatures on each report and thereby *attests* each report, and passes every attested report individually to the transmission protocol. Details of the report attestation protocol are shown in Algorithm 7–8 and described next.

The outcome generation protocol passes a sequence number, certified outcome pair (sn, CO) to the report attestation protocol through a *committedOutcome*(sn, CO) event. The certified outcome CO is the committed outcome O along with the commit quorum certificate for O, i.e., a BFT quorum of distinct signed COMMIT messages for O from the same epoch.

When some oracle p_i receives CO from the outcome generation protocol, it converts it into a vector of reports R by calling the *reports*(sn, O) plugin function. It then *signs each report* in the array and sends a $[\text{REPORT-SIGS}, sn, \Sigma]$ message to all oracles, where Σ is the vector of report signatures.

When an oracle p_i has received $f + 1$ REPORT-SIGS messages for some sequence number *sn*, it first checks if it has the certified outcome for *sn* locally. If not, it asks other oracles for it as follows. Oracle p_i creates a random permutation of the $f + 1$ oracles that have sent a REPORT-SIGS message for *sn* and starts asking them one by one for the certified outcome for *sn*. It repeats this step periodically, every $\Delta_{\text{req\_cert\_commit}}$, by sending one $[\text{CERTIFIED-COMMIT-REQ}, sn]$ at a time, until it receives the certified outcome. The randomized order serves to balance the load among the oracles. When a correct oracle p_j receives a $[\text{CERTIFIED-COMMIT-REQ}, sn]$ message from p_i, it sends to p_i the corresponding certified outcome if it has not already done so. Notice that every set of the $f + 1$ oracles includes at least one correct oracle p_c. Therefore, p_i will eventually obtain the certified outcome from p_c or through a *committedOutcome*(sn, CO) event.

When the certified outcome CO for *sn* is locally available at p_i together with $f + 1$ valid REPORT-SIGS for each report r in the vector of reports derived from CO, then p_i creates an *attestation* for the report. The attested report includes, along with r, the sequence number *sn* of the corresponding outcome O, the position of r in the vector of reports as returned by *reports*(sn, O), the vector of $f + 1$ signatures for r and the vector of the oracle identities that produced the $f + 1$ signatures. The protocol then invokes the transmission protocol with the attested report via an *attestedReport* event.

The outcome generation protocol guarantees that at least $\lceil \frac{n+f+1}{2} \rceil - f \geq f + 1$ correct oracles commit an outcome for each sequence number. Therefore, it is guaranteed that eventually at least $f + 1$ oracles will send a valid REPORT-SIGS message for each sequence number and every correct oracle will invoke the transmission protocol for all reports obtained from the outcome committed to every sequence number.

Note that though outcomes are already signed in PREPARE and COMMIT messages during the outcome generation protocol, we opt to signing each report that corresponds to the outcome separately for two reasons. First, the outcome generation protocol requires $\lceil \frac{n+f+1}{2} \rceil$ oracles to sign each PREPARE and COMMIT message in order to preserve consistency. However, using attestations with $\lceil \frac{n+f+1}{2} \rceil$ signatures is more expensive in space consumption and verification time than with the $f + 1$ signatures gathered by the report attestation protocol. Second, splitting the outcome into separate reports allows for transmitting each report in a dedicated transaction, which gives applications more flexibility.

There is a trade-off between the cost of attestation verification per report and the flexibility of transmitting each report in a single transaction. Therefore, the *reports*$(\_, \_)$ plugin function may also choose to batch multiple logically related reports into a single one. This is the reason why the conversion of an outcome to a vector of reports is configurable.

6.5 Transmission

The transmission protocol forms the interface between the Offchain Reporting Protocol and the blockchain running the smart contract C. This protocol also runs continuously and concurrently to the other protocols.

Algorithm 9 is responsible for transmitting a report a resulting from an outcome produced by Algorithm 3–6 and attested in Algorithm 7–8 to C. It receives one *attestedReport*(a) event for every report a attested by the attestation protocol (Algorithm 7–8) and then creates a suitable transaction containing a and submits this to C. Under ideal conditions, the report attestation algorithm starts the transmission protocol at roughly the same time across all oracles.

The algorithm first filters incoming attested reports to avoid redundant transmissions and to reduce gas

costs. In particular, we aim to protect against a scenario where many similar reports are produced in quick succession. In such a case, we only want to transmit the first such report and discard the following ones. This can be specified for a reporting protocol through two plugin functions *should-accept-attested-report*(a) and *should-transmit-accepted-report*(a), where a denotes an attested report resulting from

The first filter, *should-accept-attested-report*(a), is invoked immediately after a has been submitted to the transmission protocol. The second filter, *should-transmit-accepted-report*(a), gives the protocol another chance to save transaction fees. A plugin may use this, for instance, to check whether C has already recorded a and made the transmission obsolete. This check is called immediately before each transmission.

To actually transmit the reports, 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. The transmitting oracles are determined by a pseudorandomly chosen permutation π of $\{p_1, \ldots, p_n\}$, which ensures that each oracle is chosen at most once as a transmitter.

In the transmission schedule, 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 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 should be such that periods in which reports with different sequence numbers should be transmitted may overlap substantially, i.e., a report with a higher sequence number may arrive before many oracles had a chance to transmit the previous one.

The value of Δ_{stage} must be set with respect to all other timeout values and in accordance with the block interval on the blockchain running C. Typically it is a multiple of the block interval on the chain. See Section 6.6 for a discussion of the timing parameters.

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 blockchain within Δ_{stage} time. This assumes that (1) miners are actively mining the blockchain 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., when the blockchain is severely congested. 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 pseudorandom function $F_x : \{0,1\}^* \to \mathrm{Sym}(n)$, where x is a secret seed known only to oracles and $\mathrm{Sym}(n)$ is the set of permutations of $\{1, \ldots, n\}$. Given the (implicit) protocol identifier, report sequence number, and position, F_x deterministically derives a permutation of the node set. As with the *leader* function described in Section 6.2, the seed x should not be known to the oracles before they are committed to their indexing in $\{p_1, \ldots, p_n\}$, 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.

The defense against this behavior is purely economical. The owner should monitor the transactions sent to 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 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.

6.6 Implementation considerations

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.

Summary of constants. We provide a unified summary of the constants used in the protocol description and analysis.

Algorithm 8 Report attestation protocol, executed by oracle p_i (part 2)

upon $\exists sn$ **s.t.** $\left|\{p_j \in \mathcal{P} | \mathit{reportSigMsgs}[sn][j] \neq \bot\}\right| = f + 1 \wedge \mathit{reports}[sn] = \bot \wedge sn \notin \mathit{missingOutcomes}$ **do**
  for $p_j \in \mathit{random\text{-}perm}\Big(\{p_j \in \mathcal{P} \big| \mathit{reportSigMsgs}[sn][j] \neq \bot\}\Big)$ **do**
    $\mathit{oraclesWithCertOutcomes}.\mathit{enqueue}(p_j)$
  $\mathit{missingOutcomes} \leftarrow \mathit{missingOutcomes} \cup \{sn\}$
  schedule $\mathit{missingOutcome}(sn)$ at *now*

upon event $\mathit{missingOutcome}(sn)$ **s.t.** $\mathit{reports}[sn] = \bot$ **do**
  send message [CERTIFIED-COMMIT-REQ, *sn*] to $\mathit{oraclesWithCertOutcomes}.\mathit{dequeue}()$
  schedule $\mathit{missingOutcome}(sn)$ at $\mathit{now} + \Delta_{\text{req\_cert\_commit}}$

upon receiving message [CERTIFIED-COMMIT-REQ, *sn*] from p_j **s.t.** $\mathit{availableCertOutcomes}[sn] \neq \bot$
    $\wedge\, p_j \notin \mathit{sentCertCommit}[sn]$ **do**
  $\mathit{sentCertCommit}[sn] \leftarrow \mathit{sentCertCommit}[sn] \cup \{p_j\}$
  send message [CERTIFIED-COMMIT, *sn*, $\mathit{availableCertOutcomes}[sn]$] to p_j

upon receiving message [CERTIFIED-COMMIT, *sn*, *CO*] from p_j **s.t.** $\mathit{availableCertOutcomes}[sn] = \bot$ **do**
  $(\mathit{outcome}, QC) \leftarrow CO$
  if $\exists ?\, e$ **s.t.** $\left|\left\{p_j \in \mathcal{P} \;\middle|\; QC[j] = [\text{COMMIT}, e, sn, \sigma] \wedge \mathit{verify}_j\big(\text{COMMIT} \| e \| sn \| \mathit{H(outcome)}, \sigma\big)\right\}\right| \geq BQ(n, f)$ **then**
    $\mathit{availableCertOutcomes}[sn] \leftarrow CO$ // may trigger sending a REPORT-SIGS message

Algorithm 9 Transmission protocol (executed by every oracle p_i).

state
  timer T_{transmit}, initially stopped: delays until next report should be transmitted

upon event $\mathit{attestedReport}(a)$ **do**
  $(sn, pos, \_\_, \_\_, \_\_) \leftarrow a$
  if $\mathit{should\text{-}accept\text{-}attested\text{-}report}(a)$ **then**
    $\Delta_{\text{transmit}} \leftarrow \mathit{transmit\text{-}delay}(i, sn, pos)$
    schedule $\mathit{scheduledReport}(a)$ at $\mathit{now} + \Delta_{\text{transmit}}$

upon event $\mathit{scheduledReport}(a)$ **do**
  if $\mathit{should\text{-}transmit\text{-}accepted\text{-}report}(a)$ **then**
    send blockchain transaction with a to C

function $\mathit{transmit\text{-}delay}(i, sn, pos)$
  $\pi \leftarrow F_x(sn \| pos)$ // derive pseudorandom permutation of $\{1, \ldots, n\}$
  $k \leftarrow k$ such that $\sum_{j=1}^{k-1} s_j < \pi(i) \leq \sum_{j=1}^{k} s_j$ // assuming $s_0 = 0$
  return $k \cdot \Delta_{\text{stage}}$

Global

n is the number of oracles. The current implementation assumes $n \leq 31$.

Δ is the assumed upper bound on communication latency during periods of synchrony. The choice of all other time constants is constrained by this. In typical WAN deployments, this value is on the order of hundreds to thousands of milliseconds. Note that Δ plays a role for the analysis but is never explicitly used as a configuration parameter.

Pacemaker

$\Delta_{\textbf{process}}$ denotes a maximum on the processing delay of any function implemented by a reporting plugin.[1] In practice, this value is typically on the order of tens to hundreds of milliseconds.

$\Delta_{\textbf{progress}}$ is the time during which a leader of an epoch e must achieve progress or be replaced. The outcome generation protocol defines "progress" as periodically committing an outcome, indicated by *progress* events, or aborting e. After GST and with a correct leader, outcome generation must ensure progress and the pacemaker monitors this by observing *progress* events. Recall that every correct oracle triggers *progress* whenever it finishes a round and commits an outcome.

In particular, from the moment when the first correct oracle initializes epoch e, the leader of the epoch must have enough time to initialize e, make $2f + 1$ correct oracles commit the outcome of the first round of e, and trigger *progress*. Subsequently, the leader is supposed to start a new round after each interval of Δ_{round}. As one round might finish quickly and a next round might exhaust its maximal duration, two *progress* events might occur up to $2 \max\{\Delta_{\text{round}}, \Delta_{\text{grace}}\}$ apart. Consequently, Δ_{progress} must be set to $2 \max\{\Delta_{\text{round}}, \Delta_{\text{grace}}\}$ or a larger value. This also gives enough time for the first round of an epoch to start.

$\Delta_{\textbf{resend}}$ is the interval at which nodes resend NEW-EPOCH-WISH messages.

Atomic outcome generation

$\Delta_{\textbf{initial}}$ is the time during which, after GST, a leader must ensure that $n - f$ oracles start the first round of outcome generation or be replaced. The Δ_{initial} parameter must therefore be set to a value such that form the time that the first correct oracle initializes epoch e, the leader has enough time to initialize e and gather $\lceil \frac{n+f+1}{2} \rceil$ EPOCH-START-REQ messages and for all correct oracles to receive the corresponding EPOCH-START message from the leader. Moreover, Δ_{initial} should be shorter than Δ_{progress}, as this allows oracles to terminate an epoch quickly in case the leader is not responsive right at the beginning of the epoch.

$\Delta_{\textbf{round}}$ is the minimum waiting time of the leader to start a new round. This value is useful for limiting the speed at which rounds progress, e.g., to enable more precise control of the overall resource consumption. It may be set to a zero or to a small value, such that a next round starts immediately after the conclusion of the previous one.

$\Delta_{\textbf{grace}}$ is the duration of the grace period during which observations of delayed oracles are still considered by the leader, even after it has gathered enough of them. This value is useful in cases where, in the happy path, one wants to give more oracles than strictly needed an opportunity to contribute observations to the outcome. This parameter sets a minimum period for the time that one round takes to complete, i.e., in situations with instantaneous message delivery. If $\Delta_{\text{round}} \leq \Delta_{\text{grace}}$ then successive rounds may also start without a delay in between.

ρ is the maximum number of rounds in an epoch.

Transmission

[1] In the implementation, a separate maximum processing delay can be configured for *each* function of a reporting plugin. This flexibility is not needed for the analysis in this document, however.

Constant	Value
Δ_{process}	50 milliseconds
Δ_{progress}	2 seconds
Δ_{resend}	5 seconds
Δ_{initial}	500 milliseconds
Δ_{round}	250 milliseconds
Δ_{grace}	50 milliseconds
ρ	10 rounds
Δ_{stage}	2 seconds

Table 1. Example values for constants for the reporting plugin for numerical values. The table assumes a Δ of 150 milliseconds. We assume that we are running over a WAN with high-quality networking between oracles and that the target blockchain produces multiple blocks per second.

$\Delta_{\textbf{stage}}$ is used to stagger stages of the transmission protocol. Multiple blocks should be mineable on the blockchain hosting C in this period.

If Δ_{round} and Δ_{grace} are both set to 0, then the outcome generation protocol is (optimistically) responsive [PS17], as it proceeds as fast as the network speed permits.

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

Example values for constants. Table 1 shows a realistic example configuration for the various constants used to configure the protocol.

7 Reporting plugin for numerical values

This reporting plugin supports the median of prices or other numerical values. 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. The range of blocks to calculate the median is such that any two reports with sequence numbers *sn* and *sn* + 1 contain median prices for contiguous ranges of blocks. The range of blocks refers to the blockchain B on which contract C runs. Put differently, for each block height in B, there exists a unique report.

The implementation of the plugin is described in Algorithm 10. We provide a summary of its functionality grouped by subprotocol invoking the plugin.

Atomic outcome generation. Each observation includes an observed value d, the timestamp t of the observation, and the height b of the tip of the blockchain B when the observation is made. The *valid-observation*(__) function ensures that the observations are well-formed. A malformed observation indicates a faulty oracle; therefore, such observations should be ignored.

For reporting numbers from a totally ordered domain, the reporting plugin uses a median computation in the *outcome*(__) function. In this way, it prevents Byzantine oracles from significantly affecting the reported values by introducing too big or too small values in the observation set. Not only the observed value (d) in a

Algorithm 10 Reporting plugin implementation for numerical (ordered) values (executed by every p_i).

function *query*(O_{prev}, sn)
  return $\perp$

function *observation*(O_{prev}, sn, Q)
  $t \leftarrow$ *now*() // the current local time at p_i
  $d \leftarrow$ *value*() // the current realization at p_i of the reported (data) value
  $b \leftarrow$ *blockheight*() // the height of current tip of the blockchain
  return (t, d, b)

function *valid-observation*($O_{\text{prev}}, sn, Q, v$)
  return *valid*(v) // *valid*() checks if the observation is well-formed

function *observation-quorum*(O_{prev}, sn, Q)
  return $2f + 1$

function *outcome*($O_{\text{prev}}, sn, Q, B$)
  $t \leftarrow$ *median*($\{t' | (\_\_, (t', \_\_, \_\_)) \in B\}$) // recall B is a vector of index-observation pairs
  $d \leftarrow$ *median*($\{d' | (\_\_, (\_\_, d', \_\_)) \in B\}$)
  if $sn = 1$ **then** // "genesis" outcome
    $(\_\_, \_\_, b'_{\text{start}}, b'_{\text{end}}, \_\_) \leftarrow (0, 0, 0, 0, 0)$
  else
    $(\_\_, \_\_, b'_{\text{start}}, b'_{\text{end}}, \_\_) \leftarrow O_{\text{prev}}$
  if $b'_{\text{start}} \leq b'_{\text{end}}$ **then**
    $b_{\text{start}} \leftarrow b'_{\text{end}} + 1$
  else
    $b_{\text{start}} \leftarrow b'_{\text{start}}$
  $b_{\text{end}} \leftarrow$ *median*$\big(\{b | (\_\_, (\_\_, \_\_, b)) \in B\}\big)$
  shouldreport $\leftarrow (b_{\text{start}} \leq b_{\text{end}})$
  $O \leftarrow (t, d, b_{\text{start}}, b_{\text{end}}, \textit{shouldreport})$
  return O

function *reports*(sn, O)
  $(t, d, b_{\text{start}}, b_{\text{end}}, \textit{shouldreport}) \leftarrow O$
  if *shouldreport* **then**
    return $[(t, d, b_{\text{start}}, b_{\text{end}})]$ // a vector with a single report
  else
    return $[\perp]$ // an empty vector

function *should-accept-attested-report*(a)
  $(\_\_, \_\_, r, \_\_, \_\_) \leftarrow a$
  $(\_\_, \_\_, \_\_, b_{\text{end}}) \leftarrow r$
  return $\neg$*report-for-height*(b_{end})

function *should-transmit-accepted-report*(a)
  $(\_\_, \_\_, r, \_\_, \_\_) \leftarrow a$
  $(\_\_, \_\_, \_\_, b_{\text{end}}) \leftarrow r$
  return $\neg$*report-for-height*(b_{end})

function *report-for-height*(h)
  // Queries the smart contract C and returns TRUE if and only if there exists a report for block height h.

report is obtained as the median of observations, also the timestamp (t) and the block height (b) are calculated as medians. In particular, the *observation-quorum*(__) is set to $2f + 1$, such that for d, t, and b, respectively, the corresponding median lies in an interval of two values that have been observed by correct oracles.

The *outcome*(__) function also ensures that the range of blocks $\{b_{\text{start}}, \ldots, b_{\text{end}}\}$ covered by the report directly follows the range of blocks from the previous outcome $\{b'_{\text{start}}, \ldots, b'_{\text{end}}\}$, i.e., such that $b_{\text{start}} = b'_{\text{end}} + 1$. Moreover, the upper limit b_{end} of the block range in the current report should be at least equal to the lower limit b_{start}. If the median b of the reported block numbers of the blockchain tip is greater than or equal to b_{start}, then b_{end} is assigned b. If b is less than b_{start}, then the plugin function indicates that no report should be generated for this round. This is captured by the *shouldreport* flag, which is a field of the outcome.

Report attestation. The *reports*(__) function transforms the committed outcome into a vector of reports. Each outcome is transformed into a vector of at most one report, according to the *shouldreport* flag.

Transmission. The two plugin functions *should-accept-attested-report*(__) and *should-transmit-accepted-report*(__) are relevant for transmission. Recall that *should-accept-attested-report*(__) performs its check whenever the transmission protocol is invoked and *should-transmit-accepted-report*(__) performs its check right before the oracle transmits the attested report.

Both functions aim to prevent that a report r for block range $\{b_{\text{start}}, \ldots, b_{\text{end}}\}$ is transmitted to smart contract C if any report for some block height in $\{b_{\text{start}}, \ldots, b_{\text{end}}\}$ has already been finalized on blockchain B. Since the protocol guarantees that no two different attested reports have overlapping block ranges, it suffices to check that no report is finalized in B for the upper limit of the block range b_{end}. To that end, both plugin functions query contract C.

8 Analysis

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

Throughout this section, we assume that the local processing time of any correct oracle after receiving one or multiple messages is at most ϵ, which is small compared to all other durations under consideration:

ϵ is an upper bound on the processing latency of any message and local event between receiving and sending messages over the network. It includes receiving multiple messages of the same type concurrently and also subsumes all processing by reporting-plugin functions.

The partially synchronous system model [DLS88] 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. Since each oracle takes no longer than ϵ to process a message, this means that during synchronous periods, i.e., after GST, all messages between correct nodes are delivered timely:

Δ is the upper bound on the time taken by the network to deliver a message sent from one correct oracle to another one.

For understanding the arguments in this section, it is also important to recall that all oracles communicate by sending messages to each other using reliable and authenticated point-to-point links.

8.1 Pacemaker

The pacemaker protocol (Section 6.2) runs continuously. It interacts with a connected outcome generation protocol by consuming (input) events of type *progress* and *newEpochReq* and by emitting (output) events of type *newEpochStart*(e, ℓ). The latter indicate that the outcome generation protocol should start epoch e with leader ℓ.

***progress*:** Originates from the outcome generation protocol and indicates to the pacemaker that the leader has made progress, i.e., an outcome has been committed.

***newEpochReq*:** Originates also from the outcome generation protocol and indicates a leader change to the pacemaker. This signals either that the current epoch has reached its maximal number of rounds or that the leader does not make progress in the current epoch, according to the oracle's local view and clock; therefore, the oracle should advance to the next epoch and a new leader.

***newEpochStart*(e, ℓ):** Signals to the outcome generation to start epoch e with p_ℓ as leader.

The pacemaker protocol internally uses a timer T_{progress} with duration Δ_{progress} that captures the assumption that the outcome generation protocol should repeatedly emit *progress* or *newEpochReq* events that are no more than Δ_{progress} apart. Since every epoch of the outcome generation protocol is driven by a leader oracle p_ℓ, we say that the *oracle times out on* p_ℓ when no *progress* event occurs between two successive timeouts from T_{progress}. We say than an oracle *aborts* epoch e when it times out on the epoch leader or when it receives a *newEpochReq* from the outcome generation protocol.

Furthermore, the primitive assumes from the connected outcome generation protocol that whenever a correct oracle triggers an event *newEpochStart*(e, ℓ), then the oracle later either times out on p_ℓ or issues a *newEpochReq* event. This is needed to ensure the first property in the following definition.

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

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

Eventual succession: After GST, if more than f correct oracles abort epoch e, then all correct oracles will start epoch $e + 1$ after at most $2\Delta + 2\epsilon$.

Putsch resistance: A correct oracle that has last started epoch e does not start an epoch $e' > e$ unless at least one correct oracle has aborted epoch e.

Monotonicity: If a correct oracle starts some epoch e and later starts an epoch e', then $e' > e$.

We now show that Algorithm 2 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. The lemma mentions that a correct oracle that has last started a particular epoch e may broadcast a NEW-EPOCH-WISH message with an epoch $e' > e$. According to the protocol, an oracle does this if and only if also aborts epoch e; this connects the condition to the *putsch resistance* property of the pacemaker.

Lemma 1. *Suppose the maximal epoch that has been started by any correct oracle is $\bar{e}$ and let $\overline{ne}$ denote the maximal value of variable ne at any correct oracle. If no correct oracle aborts $\bar{e}$, then*

(a) no correct oracle broadcasts $[\text{NEW-EPOCH-WISH}, e']$ *with* $e' > \overline{ne}$*;*

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

Proof. For claim (a), let's assume the contrapositive, i.e. there exists some correct oracle which broadcast $[\text{NEW-EPOCH-WISH}, e', \_\_]$ with $e' > \overline{ne}$. Let p^* be the first correct oracle that sends $[\text{NEW-EPOCH-WISH}, e', \_\_]$ with $e' > \overline{ne}$. According to Algorithm 2, for p^* it holds that $ne = e' > \overline{ne}$. Also according to Algorithm 2, a correct oracle, and thus p^*, can advance *ne* either independently by aborting epoch $\bar{e}$, or else can advance *ne* via the new-epoch-wish messages. The first case is excluded by the lemma hypothesis, thus, we now focus on the latter case. The Algorithm 2 requires for p^* to have received $[\text{NEW-EPOCH-WISH}, e_s, \_\_]$ messages from oracles p_s, where $s \in S$, for a set $S \subset [n]$ of cardinality at least $f + 1$, where $e_s \geq ne > \overline{ne}$. Any such set must contain at least one i such that p_i is a correct oracle, which contradicts the assumption that p^* was the first correct oracle that sent such a message. Therefore, the first correct oracle that sends such a message, must have aborted $\bar{e}$, a contradiction.

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 NEW-EPOCH-WISH message containing epoch $\bar{e} + 2$ or higher, through the "NEW-EPOCH-WISH 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 $\bar{e} + 1$. □

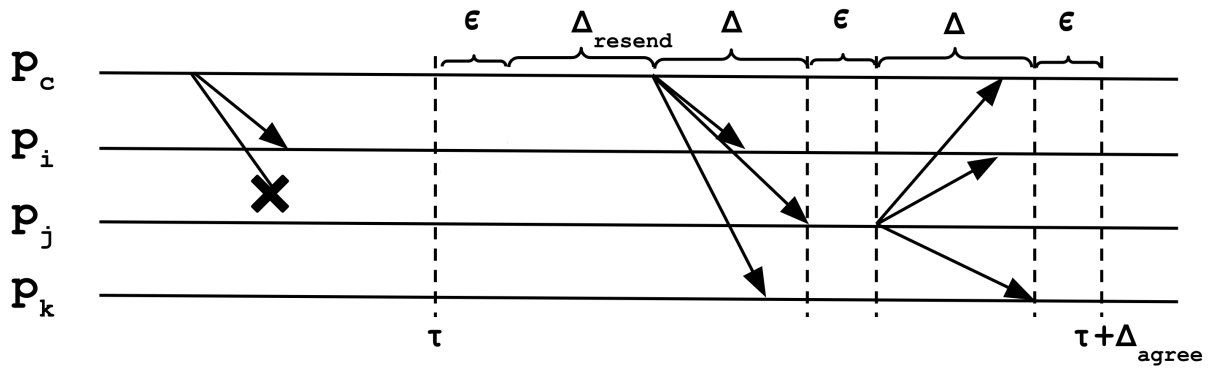


Figure 4. Oracle p_i is the first to enter epoch e upon receiving $2f + 1$ NEW-EPOCH-WISH from at least $f + 1$ correct oracles. One of them, p_c, crashed while sending the NEW-EPOCH-WISH. It recovers the latest after GST and resends it to all oracles. Oracle p_j represents of the correct oracles that receive the $f+1$ NEW-EPOCH-WISH messages and because of "NEW-EPOCH-WISH amplification rule" sends its own NEW-EPOCH-WISH. Oracle p_k represents a correct oracle that receives the $2f + 1$ NEW-EPOCH-WISH messages, the latest thanks to amplification.

The next lemma establishes a condition for satisfying eventual agreement, i.e., that all correct oracles start the same epoch at some time after GST and remain in this epoch long enough.

Lemma 2. *Consider an epoch e such that no correct oracle has started any higher epoch than e. If some correct oracle has started e before GST, then define a point in time τ to be GST; otherwise, let τ be the time when the first correct oracle started 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 $\Delta_{agree} = \Delta_{resend} + 2\Delta + 3\epsilon$ after τ, then all correct oracles have started 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) may differ between correct oracles because of crashes. In particular, correct oracles that have recovered may have missed arbitrarily many messages.

Let p_i be the oracle that has started epoch e at time τ according to the assumption. Observe that τ is GST or later. Oracle p_i has received more than $2f$ NEW-EPOCH-WISH messages containing an epoch value of at least e. More than f of those messages were sent by correct oracles.

If some of these f correct oracles had crashed while sending their NEW-EPOCH-WISH message and meanwhile recovered, they 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 NEW-EPOCH-WISH messages with an epoch value of e or higher at the latest by $\Delta_{\text{resend}} + \epsilon$ after τ.

Thus, after at most $\Delta_{\text{resend}}+\Delta+2\epsilon$, every correct oracle has received and processed more than f NEW-EPOCH-WISH messages containing an epoch value of at least e. According to the algorithm and the "NEW-EPOCH-WISH amplification rule", since epoch values of at least e are reported more than f times, every correct oracle then also broadcasts a NEW-EPOCH-WISH message containing epoch e or higher by this time, i.e., not longer than $\Delta_{\text{resend}} + \Delta + 2\epsilon$ after τ.

These messages have been received and processed by all correct oracles after at most $\Delta_{\text{resend}} + 2\Delta + 3\epsilon = \Delta_{\text{agree}}$. Hence, every correct oracle has received $n - f > 2f$ NEW-EPOCH-WISH messages with an epoch at least e and has started epoch e after Δ_{agree}. Figure 4 visualizes how some correct oracle p_k starts epoch e the latest by $\tau + \Delta_{\text{agree}}$. Lemma 1(a) implies that no oracle starts a higher epoch afterwards, unless some correct oracle indicates a leader change in epoch e. □

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 started 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 start epoch $\overline{ne}$ by sending NEW-EPOCH-WISH 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 $\overline{ne}$.

Moreover, because a correct oracle eventually ends every epoch it has started, it also does so after GST. This implies that an unbounded sequence of epochs is started after GST. Due to the implementation of the *leader*(e) function that selects all oracles approximately equally, also correct oracles become leaders for an unbounded number of epochs.

We can summarize this behavior less precisely in the following lemma.

Lemma 3 (Eventual agreement). *Eventually, every correct oracle has initialized the same epoch e with a correct leader and does not abort it unless some correct oracle times out or indicates a leader change.*

The next lemma addresses liveness of the pacemaker after the network has stabilized.

Lemma 4 (Eventual succession). *After GST, if more than f correct oracles abort epoch e, then all correct oracles will advance to epoch $e + 1$ after at most $2\Delta + 2\epsilon$.*

Proof. First, observe that when a correct oracle aborts epoch e, it sets $ne = e + 1$. Let τ be the point in time when the $(f + 1)^{st}$ correct oracle aborts epoch e. Notice that the statement only considers the time after GST, when no more oracles crash. All correct oracles therefore receive $f + 1$ NEW-EPOCH-WISH messages for epoch $e + 1$ by time $\tau + \Delta$ and because of the "NEW-EPOCH-WISH amplification rule" send their own NEW-EPOCH-WISH messages for epoch $e + 1$ by time $\tau + \Delta + \epsilon$. Therefore, by time $\tau + 2\Delta + 2\epsilon$, all correct oracles have received and processed at least $2f + 1$ NEW-EPOCH-WISH messages for epoch $e + 1$. Hence, all correct oracles have started epoch $e + 1$ after at most $2\Delta + 2\epsilon$. □

Lemma 5 (Putsch resistance). *A correct oracle does not send a* NEW-EPOCH-WISH *message for an epoch $e' > e$ unless at least one correct oracle aborts epoch e.*

Proof. Let p_i be a correct oracle that sends a NEW-EPOCH-WISH message. We distinguish two cases. Either p_i has aborted epoch e or p_i has not aborted epoch e. In the first case, the lemma is satisfied trivially, as the correct node which aborts is p_i itself.

Otherwise, p_i has received $2f + 1$ NEW-EPOCH-WISH messages for epochs higher than e, of which at least $f + 1$ are from correct oracles. Notice that $\overline{ne}$ in Lemma 1 is the maximal epoch value that any correct oracle has ever sent in a NEW-EPOCH-WISH message. Assume that e is the highest epoch started by any correct oracle, hence we have $\overline{ne} = e$.

Towards a contradiction, suppose that some correct oracle has sent a NEW-EPOCH-WISH message for epoch e', yet no correct oracle has aborted epoch e.

According to the algorithm, this means the correct oracle has received NEW-EPOCH-WISH messages for some epoch higher than e from more than $2f$ oracles, of which some must have been sent by correct oracles. But by Lemma 1(a) with $\overline{ne} = e$, no correct oracle has sent a NEW-EPOCH-WISH message with an epoch higher than e under the assumptions of putsch resistance. The lemma follows. □

Lemma 6 (Monotonicity). *If a correct oracle starts some epoch e and later starts an epoch e', then $e' > e$.*

Proof. Every oracle stores its current epoch in variable e. According to the condition within the "agreement rule" and the computation of $\overline{e}$, variable e changes only to $\overline{e} > e$. Furthermore, the oracle always includes its local value e in the event that starts a new epoch. This establishes that successively started epochs increase monotonically. □

The statements of Lemmas 3–6 imply all properties of a pacemaker protocol. Therefore, the following result summarizes our analysis.

Theorem 7. *Algorithm 2 implements a pacemaker protocol.*

8.2 Atomic outcome generation

We now give a definition for outcome generation. As illustrated in Section 6.3, this protocol receives *newEpochStart*(e, ℓ) events from the pacemaker protocol. It emits back *progress* and *newEpochReq* events. The former indicates that the oracle committed an outcome and the latter indicates that the pacemaker protocol should move to a new epoch. Moreover, the protocol emits *committedOutcome*(sn, CO) events destined for the report attestation protocol to indicate that the outcome generation protocol committed a certified outcome CO to sequence number *sn*. Formally:

newEpochStart(e, ℓ)**:** Instructs the oracle to start epoch e and elects p_ℓ to be leader.

***progress*:** Indicates that the outcome generation protocol has made progress by committing an outcome. This implies that the most recently elected leader p_ℓ is not suspected as faulty.

***newEpochReq*:** Indicates that the outcome generation protocol requests a switch to a new epoch and aborts the current epoch. This may occur either because the T_{initial} timer expired at the beginning of the epoch or because the epoch has committed the maximum foreseen number of outcomes.

committedOutcome(sn, CO)**:** Indicates that the oracle in the outcome generation protocol committed a (certified) outcome CO to sequence number *sn*. Recall that a certified outcome CO is a tuple that contains an outcome returned by $\mathit{outcome}(\_\_)$ and a commit-quorum certificate.

As the reporting mechanism is configurable through reporting plugins according to Section 5, oracles call $\mathit{observation}(O_{\text{prev}}, sn, Q)$ for obtaining an *observation value* v, which may change for every such call. When this function is invoked, we say that an oracle *observes value v in context O_{prev}, with sequence number sn, and for query Q*. Here and in the following, O_{prev} denotes most recently committed outcome, which is found in the *outcome* variable.

Whenever an oracle p_i receives or emits an event h, such as committing an outcome, and h occurs after p_i starts some epoch e and before it starts any epoch higher than e, we abbreviate this by saying *event h occurs at p_i in epoch e*.

Intuitively, the outcome generation protocol commits certified outcomes to sequence numbers in the same order at all correct processes. It is therefore close to a primitive called *atomic broadcast* or *total-order broadcast* [CGR11], which outputs a global, totally ordered sequence of messages. (In blockchain contexts, this primitive is very often simply known as *consensus*.) However, outcome generation is weaker: it maintains the total order and commits an outcome to every sequence number, but not every correct oracle commits the complete sequence of outcomes, rather the primitive requires only that for every sequence number at least $f+1$ correct oracles commit the outcome. This is apparent from the *weak validity* property in the definition below and suffices for the intended use of outcome generation.

In the following, let

$$\Delta^+ = 7\epsilon + 5\Delta + \Delta_{\text{grace}}$$

and also assume $\Delta_{\text{round}} > \Delta^+$. Note this implies also $\Delta_{\text{round}} > \Delta_{\text{grace}}$. The protocol actually permits also $\Delta_{\text{round}} < \Delta_{\text{grace}}$, for instance with $\Delta_{\text{round}} = 0$, which means that a new round may start immediately after the leader completes the previous one. However, since it complicates the liveness analysis, we make this simplifying assumption here.

Definition 2. An *outcome generation* protocol satisfies these conditions:

Integrity: If a correct oracle p_i commits an outcome O for sequence number *sn* in epoch e, then (1) $O = \mathit{outcome}(O_{\text{prev}}, sn, Q, B)$ and the leader $p_{\ell'}$ of some epoch $e' \leq e$ has sent Q as query in a ROUND-START message with sequence number *sn*, and (2) B is a vector as follows: for a set P of at least $\mathit{observation\text{-}quorum}(O_{\text{prev}}, sn, Q) - f$ distinct correct oracles p_j, it holds that p_j has observed value $B[j]$ in context O_{prev} and with sequence number *sn*. Moreover, if $p_{\ell'}$ is correct, then $Q = \mathit{query}(O_{\text{prev}}, sn)$.

No duplication: No correct oracle commits multiple outcomes for the same sequence number.

Consistency: No two correct oracles commit different outcomes for the same sequence number.

Completeness: If some correct oracle commits an outcome O for a sequence number *sn*, then for each sequence number in $\{1, \ldots, sn\}$, some correct oracle has committed an outcome.

Weak totality: If a correct oracle p_i commits an outcome O for sequence number *sn*, eventually at least $f + 1$ correct oracles commit O to *sn*.

Liveness: Consider a time after GST when the leader p_ℓ of epoch e is correct:

1. Suppose leader p_ℓ either starts the epoch or invokes query at some time τ. Then p_ℓ invokes query again within Δ_{round} after τ or the epoch is aborted.
2. If p_ℓ calls query(_) with sequence number *sn* at some time τ and no correct oracle triggers *newEpochStart* within time Δ^+ after τ, then every correct oracle commits an outcome O for sequence number *sn* within Δ^+ after τ.
3. After all correct oracles have started some epoch (e, ℓ), then every correct oracle requests a switch to a new epoch or indicates that it makes progress at least once in every interval of length $\Delta_{\text{round}} + \Delta^+$.

Bounded epoch length: Let p_i be the first correct oracle that commits an outcome to some sequence number *sn* in an epoch e. Then no correct oracle commits any outcome for a sequence number larger than $sn + \rho + 1$.

The remainder of this section is devoted to a proof that Algorithm 3–6 implements an outcome generation protocol.

Lemma 8 (Integrity). *Suppose an correct oracle p_i commits an outcome O for sequence number sn in epoch e. Then:*

1. $O =$ *outcome*(O_{prev}, sn, Q, B) *and the leader $p_{\ell'}$ of some epoch $e' \leq e$ has sent Q as query in a* ROUND-START *message with sequence number sn; and*
2. *B is a vector as follows: for a set P of at least* observation-quorum$(O_{prev}, sn, Q) - f$ *distinct correct oracles p_j, it holds that p_j has observed value $B[j]$ in context O_{prev} and with sequence number sn. Moreover, if $p_{\ell'}$ is correct, then $Q =$* query(O_{prev}, sn).

Proof. Observe that for an outcome to be committed to *sn*, its hash must be contained in a commit-quorum certificate for some epoch e' and sequence number *sn*. Through the way such certificates are constructed, this hash must also be contained in a prepare-quorum certificate for *sn* in turn. According to the protocol, every correct oracle has obtained O from the deterministic function *outcome*$(O_{\text{prev}}, sn, Q, B)$, which it runs on its own O_{prev} and values sn, B from a PROPOSAL message and *query* obtained earlier, in the same round, from a ROUND-START message. The PROPOSAL and the ROUND-START messages have been sent by the leader $p_{\ell'}$ of epoch e' in round *sn*.

Recall that O_{prev} reflects the value of variable *outcome*, specifically, its content at the time when the PROPOSAL message is received. This value is the same at every correct oracle because *outcome* is either taken from the EPOCH-START message (in the first round within some epoch) or set to the outcome obtained in the previous round, where it is computed by the *outcome*(_) function. This establishes the first property.

Furthermore, when an oracle receives a PROPOSAL message with vectors B and Σ, it ensures that B contains at least *observation-quorum*(O_{prev}, sn, Q) entries with index-observations pairs (j, v_j) and the corresponding valid signatures σ_j in Σ. A correct oracle p_j only issues a signature σ_j on an OBSERVATION message if it has obtained v_j from a call to *observation*$(O_{\text{prev}}, sn, query)$, where it has taken *query* from the ROUND-START message from $p_{\ell'}$. Note that if $p_{\ell'}$ is correct, then it has set *query* $=$ *query*(O_{prev}, sn). The second property follows directly from these observations. □

The next two lemmas address the consistency of committed outcomes for a given sequence number.

Recall that within an epoch the protocol proceeds in rounds, which are identified by *sn*, and produces one prepare-quorum certificate and one commit-quorum certificate in each round. On the other hand, two rounds in different epochs may have the same sequence number. For an epoch e, sequence number *sn*, and outcome O, a prepare-quorum certificate is a collection of $BQ(n, f)$ valid signatures from distinct oracles on the string PREPARE$\|e\|sn\|H(O)$. A commit-quorum certificate has the same structure, except that the tag in the signatures is COMMIT instead of PREPARE.

Lemma 9 (In-epoch consistency). *Consider two correct oracles p_i and p_j in the same epoch e when they receive* PREPARE *or* COMMIT *messages, respectively, with the same sequence number sn.*

1. *If p_i obtains a prepare-quorum certificate (that it assigns to prepareQC) for sequence number sn and outcome O and p_j similarly obtains a prepare-quorum certificate for sequence number sn and outcome O', then $O = O'$.*

2. *If p_i obtains a prepare-quorum certificate (that it assigns to prepareQC) for sequence number sn and outcome O and p_j obtains a commit-quorum certificate (that it assigns to commitQC) for sequence number sn and outcome O', then $O = O'$.*

3. *If p_i obtains a commit-quorum certificate (that it assigns to commitQC) for sequence number sn and outcome O and p_j similarly obtains a commit-quorum certificate for sequence number sn and outcome O', then $O = O'$.*

Proof. According to the protocol every correct oracle signs at most one PREPARE message and at most one COMMIT message, respectively, for particular values of e and *sn*. This follows from the monotonicity property of the pacemaker: variable e may only increase at an oracle, and the oracle therefore does not sign two of these messages with the same e. Claims 1 and 3 then follow from the standard argument about Byzantine quorums, that any two quorums of size $BQ(n, f) = \lceil \frac{n+f+1}{2} \rceil$ overlap in at least one correct oracle, assuming that $n > 3f$.

Claim 2 connects prepare-quorum certificates to commit-quorum certificates with the same sequence numbers. According to the algorithm, an oracle only signs a COMMIT message for its current epoch e and after it has obtained a valid prepare-quorum certificate with the same e and *sn* values.

By the fact that the epoch e only increases, it follows that there can be at most one outcome in a valid prepare-quorum certificate for some e and *sn*. And since a commit-quorum certificate requires COMMIT messages signed by multiple correct oracles with the same *sn*, and Claim 1 together with the protocol implies that, for particular e and *sn*, there is a unique outcome O for which correct oracles sign COMMIT messages. □

Whenever an oracle p_i obtains a commit-quorum certificate for sequence number *sn* and outcome O, then p_i computes the corresponding certified outcome CO, consisting of O and the commit-quorum certificate, and commits CO to *sn*. This may occur when p_i starts an epoch, in response to receiving an EPOCH-START message, or within a round of an epoch when sufficiently many COMMIT messages have been received.

Lemma 10 (Consistency). *Suppose some correct oracle p_i has committed outcome O to sn and a correct oracle p_j has committed outcome O' to sn. Then $O = O'$.*

Proof. Recall that correct oracle p_i commits outcome O with sequence number *sn* only when it possesses a valid commit-quorum certificate QC that contains *sn*, O, and additionally some epoch e. Assume that p_j commits O' with epoch e' and *sn*. We distinguish three cases:

1. Case $e = e'$: Since both commit-quorum certificates originate in the same epoch, Lemma 9 implies that $O = O'$.

2. Case $e < e'$: Consider the commit-quorum certificate QC' that has caused p_j to commit O' to *sn*.

   It means there exist two epochs (e' and the smaller e) for which valid commit-quorum certificates, QC' and QC respectively, on the same sequence number *sn* exist. By quorum intersection, there exists at least one correct oracle that has signed a COMMIT that exists in both quorum certificates, which suggests that it has collected valid prepare-quorum certificates for e', sn, O' and e, sn, O respectively. Again by quorum intersection, this suggests that some correct oracle p_k has signed PREPARE messages for both e', sn, O' and e, sn, O. Recall that, according to the protocol, a correct oracle only signs a PREPARE message in epoch e' for *sn* after accepting a PROPOSAL message for *sn* or a EPOCH-START message with a valid prepare-quorum certificate of some smaller epoch e_0.

   In the first case, p_k only accepts PROPOSAL messages for epoch e' so signing a PREPARE message for both e', sn, O' and e, sn, O contradicts our assumption that $e < e'$. In the second case, the protocol ensures that p_k signs a PREPARE message in epoch e' only for the same sequence number *sn* as found in

the prepare-quorum certificate from epoch e_0. In particular, *sn* is the sequence number extracted from *highCertTimestamp* in the EPOCH-START message received from the leader. By the assumption that p_j commits O' to *sn* in epoch e' and Lemma 9 it follows that the prepare-quorum certificate in epoch e_0 is for an outcome $O'' = O'$.

It remains to prove that $O'' = O$. We show this by induction on the number of epochs between e and e_0. For the base case we have that $e_0 = e$. This suggests that there exists a prepare-quorum certificate for sn, O'' in the same epoch at which p_i committed O to *sn* and, by Lemma 9, $O'' = O$. For the induction step we assume that there exists some epoch e_n, such that $e_n - e = n$, in which there exists a prepare-quorum certificate for $O'' = O$ and *sn* is the highest prepared sequence number by any correct oracle in epoch e_n; otherwise *sn* would have been committed in epoch e_n and $e_n = e_0$. We want to prove that that if in the next epoch e_{n+1} there also exists a prepare-quorum certificate for O''' and *sn*, then $O''' = O''$. The protocol guarantees that a correct oracle in epoch e_{n+1} only signs a PREPARE message for the same outcome, sequence number, and epoch as the highest certified outcome, sequence number, and epoch in a valid EPOCH-START message. Moreover, when *sn* is the highest prepared outcome among all correct oracles in epoch e_n, there exists no valid EPOCH-START for e_{n+1} without *sn* as the sequence number of the highest certified outcome. Therefore, the only outcome that can be prepared in epoch e_{n+1} is O'' and, thus, $O''' = O'' = O$.

3. Case $e' > e$: Since the roles of p_i and p_j are symmetric, the same reasoning as in the previous case applies.

□

Lemma 11 (No duplication). *No correct oracle commits multiple outcomes for the same sequence number.*

Proof. This property follows from the protocol by observing how variables *sn* and *certSn* may change. The former tracks the current round within the epoch and the latter contains the sequence number of the most recent outcome that the oracle has committed. In particular, whenever the oracle commits an outcome after receiving a commit-quorum certificate for *sn*, it updates *certSn* to *sn*.

One can see, by inspection of the protocol, that the $sn > certSn$ guard (inside function *Less*, called when receiving an EPOCH-START message) is satisfied before this sequence number *sn* is adopted by the oracle and committed to an outcome later. Since *certSn* is only updated at the end of a round, when the aforementioned guard has been satisfied, *certSn* is monotonically increasing, thus guaranteeing the property.

□

Lemma 12 (Completeness). *If some correct oracle commits an outcome O for a sequence number sn, then for each sequence number in* $\{1, \ldots, sn\}$*, some correct oracle has committed an outcome.*

Proof. For sequence numbers committed within one epoch by a correct oracle, the logic of the protocol immediately implies the statement, as argued in the proof of the previous lemma.

Thus, we consider sequence numbers committed in different epochs. Suppose that sn' is the largest sequence number committed by any correct oracle within epoch e'. Then for at least $f + 1$ correct oracles, it holds that their variable *certSn* is at least sn' when they enter any subsequent epoch. Furthermore, the intersection property among Byzantine quorums implies that there cannot exist a valid prepare-quorum certificate for any sequence number larger than $sn' + 1$.

Hence, a leader of any higher epoch e'' may include in the *highCertTimestamp* tuple of a valid EPOCH-START message only a sequence number that is at most $sn' + 1$. But since every correct oracle during epoch e'' ensures that it also commits an outcome with this sequence number (*sn*), unless it has committed this sequence number before, the claim of the lemma follows: the range of sequence numbers committed by correct oracles is contiguous. □

Lemma 13 (Weak totality). *If a correct oracle* p_i *commits an outcome O for sequence number sn, eventually at least* $f + 1$ *correct oracles commit O to sn.*

Proof. Consistency ensures that if some correct oracle commits an outcome O to a sequence number *sn* then any other correct oracle also commits O to *sn*.

A correct oracle may commit an outcome O after receiving an EPOCH-START message or at the end of a round. In both cases, there exists a commit-quorum certificate on O, but the epoch may be aborted immediately,

without any other oracle committing *sn*. Consider the leader p_ℓ^* of some subsequent epoch. This leader may be faulty and try to cause that fewer than $f+1$ correct oracles commit *sn*. However, if any correct oracle should commit $sn+1$ in this epoch, then at least $BQ(n,f)$ oracles will also have committed *sn*. To see why, recall that an EPOCH-START message must have contained *highCertTimestamp′* and a matching valid prepare- or commit-certificate in *highQC′*:

- If it is a prepare-certificate, then the protocol starts with sequence number *sn* from *highCertTimestamp′* into the first round with phase SENT-PREPARE; should a correct oracle commit $sn+1$ in this epoch, then also $BQ(n,f)$ oracles have sent valid COMMIT messages for *sn*.
- If *highQC′* is a commit-certificate, then every oracle compares the sequence number in *highCertTimestamp′* to its *certSn* variable and commits an outcome unless it has done so earlier. The first round of the epoch starts only afterwards.

Taken together, this ensures that at least $BQ(n,f) - f \geq f+1$ correct oracles have actually committed *sn*. Otherwise, if p_ℓ^* does not cause any correct oracle to commit an outcome to $sn+1$, the protocol will not emit *progress* events and the epoch will be aborted according to the *eventual succession* property of the underlying pacemaker protocol. Eventually, an epoch with a correct leader will start and some correct oracles commit some outcome to $sn+1$. Then at least $f+1$ correct oracles have also committed an outcome to *sn*. □

Lemma 14 (Liveness). *Consider a time after GST when the leader p_ℓ of epoch e is correct:*

1. *Suppose leader p_ℓ either starts the epoch or invokes query at some time τ. Then p_ℓ invokes query again within Δ_{round} after τ or the epoch is aborted.*
2. *If p_ℓ calls query(_,_) with sequence number sn at some time τ and no correct oracle triggers newEpochStart within time Δ^+ after τ, then every correct oracle commits an outcome O for sequence number sn within Δ^+ after τ.*
3. *After all correct oracles have started some epoch (e,ℓ), then every correct oracle requests a switch to a new epoch or indicates that it makes progress at least once in every interval of length $\Delta_{round} + \Delta^+$.*

Proof. Observe that this lemma considers only the time after GST and assumes the leader p_ℓ is correct.

The first property follows directly from the protocol and the role of the timer T_{round}. In particular, recall that the grace-period timer is set to a value Δ_{grace} smaller than Δ_{round}, according to the assumption made before Definition 2. Whenever a round finishes, a correct leader p_ℓ has also indicated progress to the outcome generation protocol. Then p_ℓ waits for the round timer to expire before it sends out a new ROUND-START message. The stated bound on Δ_{grace} ensures that a round never takes longer than this to complete, assuming a point in time after GST.

To establish the second property, suppose p_ℓ calls query(_,_) at time τ. Then it also starts a new round by sending ROUND-START messages according to the protocol. After a sequential exchange of five messages within the round (ROUND-START, OBSERVATION, PROPOSAL, PREPARE, and COMMIT), every correct oracle will commit some outcome. Including processing time of ϵ in each activation, this amounts to a maximal delay of $5\Delta + 6\epsilon$. Some more time may pass, however, because p_ℓ also enters the grace period, which accounts for an additional delay Δ_{grace} plus processing once (ϵ). In total, the delay after τ is at most $7\epsilon + 5\Delta + \Delta_{\text{grace}} = \Delta^+$. Note that throughout this, we have assumed that no correct oracle triggers *newEpochStart*. In the first round, the pacemaker's *eventual succession* property guarantees that every correct oracle enters the epoch within $2\Delta + 2\epsilon$ after the leader has entered it. This allows the late-joining correct oracles to commit within Δ^+ after the leader calls query(_,_) because they will receive the messages pertaining to the round within the same time bounds as the early-joining oracles.

For the third property, notice that the protocol instructs every correct oracle to indicate progress whenever it commits an outcome. The interval to consider starts whenever the correct leader p_ℓ starts a new round because in the ideal situation of instantaneous message delivery, a correct oracle p_i may commit some outcome and also indicate progress immediately afterwards. Completing this round and starting the next one takes Δ_{round} for the leader p_ℓ.

Hence, oracle p_i receives the next ROUND-START message from p_ℓ at most $\Delta_{\text{round}} + \Delta + \epsilon < \Delta_{\text{round}} + \Delta^+$ after its most recent progress indication. If this round exceeds the maximum number of rounds in the epoch

(ρ), then p_i requests a switch to a new epoch, which establishes the first part of the conclusion. Otherwise, p_ℓ drives the round to completion and causes p_i to commit some outcome, which takes at most Δ^+ after receiving the ROUND-START message. Putting this together implies that the subsequent progress indication may arrive up to $\Delta_{\text{round}} + \Delta^+$ after the previous one. □

Lemma 15 (Bounded epoch length). *Let p_i be the first correct oracle that commits an outcome to some sequence number sn in an epoch e. Then no correct oracle commits any outcome for a sequence number larger than $sn + \rho + 1$.*

Proof. After starting epoch e, oracle p_i may commit the first outcome either upon receiving an EPOCH-START message or at the end of the first round that it completes within e. In the first case, p_i has received a valid quorum certificate from p_ℓ that contains e and *highCertTimestamp′* with a sequence number *sn*; then p_i commits the outcome to *sn*. In the second case, the sequence number of this first round is $sn + 1$. In both cases, p_i sets *firstSnOfEpoch* $= sn+1$ and never enters a round associated to sequence number larger than *firstSnOfEpoch*$+\rho$. This implies that the maximal number of outcomes committed within a given epoch is at most $\rho + 1$. □

8.3 Report attestation

The report attestation protocol, described in Algorithm 7–8, also runs continuously. It receives *committedOutcome* events from the outcome generation protocol, and outputs *attestedReport* events, which are processed by the transmission protocol.

committedOutcome(sn, CO)**:** Signals that the outcome generation protocol committed a certified outcome CO to sequence number *sn* and starts the attestation of CO. A certified outcome CO is a tuple that contains an outcome O and a commit-quorum certificate.

attestedReport(a)**:** Indicates that attested report a is to be transmitted.

Whenever it receives a *committedOutcome* event, the report attestation protocol converts every committed outcome to a list of reports, gathers an attestation on each one, which consists of $f + 1$ signatures from unique oracles, and emits an *attestedReport* event for each report to the transmission protocol. More formally, report attestation behaves as follows.

Definition 3. A report attestation protocol satisfies these conditions:

Integrity: If some correct oracle invokes *attestedReport*(a) with an attested report a that contains sequence number *sn* and a report r, then some correct oracle has committed an outcome $CO = (O, \_\_)$ to sequence number *sn*, where $r \in O$.

Totality: If a correct oracle commits an outcome $CO = (O, \_\_)$ to sequence number *sn*, then for every report r in the list returned by *reports*(sn, O), every correct oracle collects an attestation for r and eventually invokes *attestedReport* with an attested report that contains r.

Timely inclusion: Consider a point in time after GST when $f + 1$ correct oracles have committed an outcome $CO = (O, \_\_)$ to sequence number *sn*. Then every correct oracle invokes *attestedReport* for each report r in the list returned by *reports*(sn, O) within at most $(f + 1)(\Delta_{\text{req\_cert\_commit}} + \epsilon) + 3(\Delta + \epsilon)$.

Lemma 16 (Integrity). *Suppose a correct oracle invokes attestedReport(a) with an attested report a that contains sequence number sn and a report r. Then some correct oracle has committed an outcome $CO = (O, \_\_)$ to sequence number sn, where $r \in O$.*

Proof. If the oracle p_i that invokes *attestedReport*(a), where a contains r, is correct and has itself committed the outcome CO, from which r was generated, then the protocol immediately implies this property because p_i has merely stored CO locally. On the other hand, if p_i has obtained CO in a CERTIFIED-COMMIT message from some oracle, it has verified that the commit certificate within CO is valid. The *completeness* property of the outcome generation protocol then implies that some correct oracle has committed CO to *sn*. □

Lemma 17 (Totality). *If a correct oracle commits an outcome $CO = (O, \_\_)$ to sequence number sn, then for every report r in the list returned by* reports(sn, O)*, every correct oracle collects an attestation for r and eventually invokes* attestedReport *with an attested report that contains r.*

Proof. By the *weak totality* property of the outcome generation protocol, if a correct oracle commits outcome $CO = (O, \_\_)$ to sequence number *sn*, then eventually $f + 1$ correct oracles also commit CO to *sn*. Therefore, at least $f + 1$ correct oracles send a $[\text{REPORT-SIGS}, sn, \Sigma]$, where Σ is a vector such that $\Sigma[k]$ is a signature on *sn* and report $R[k]$ in the vector of reports, $R =$ reports(sn, O).

It follows that each correct oracle receives $f + 1$ signatures for each report $R[k]$. We distinguish two cases: either a correct oracle p_i has already committed the outcome O or not. The first case also implies that p_i has the outcome O available locally and can, therefore, produce the vector of reports and verify the signatures in the REPORT-SIGS messages. Once it has collected the $f + 1$ respective signatures, it may produce an attested report a from r. The oracle then invokes *attestedReport*(a).

In the second case, the $f + 1$ signatures on *sn* indicate that at least one correct oracle q has committed an outcome for *sn*, form which p_i can eventually fetch the outcome and its commit certificate via a request in a CERTIFIED-COMMIT-REQ and receiving a response in a CERTIFIED-COMMIT message. Then p_i can generate the vector of reports R and finally invoke *attestedReport*(R) event. □

Lemma 18 (Timely inclusion). *Consider a point in time after GST when $f + 1$ correct oracles have committed an outcome O to sequence number sn. Then every correct oracle invokes* attestedReport *for each report r in the list returned by* reports(sn, O) *within at most $(f + 1)(\Delta_{\text{req\_cert\_commit}} + \epsilon) + 3(\Delta + \epsilon)$.*

Proof. Assume that $f + 1$ correct oracles have committed outcome O to sequence number *sn* at some time τ. According to the protocol, every correct oracle has therefore received and processed $f + 1$ REPORT-SIGS messages at time $\tau + \Delta + \epsilon$.

We distinguish again two cases: some correct oracle p_i has either already committed the outcome O at this point in time or it has not. In the first case, p_i can immediately invoke *attestedReport* for all reports that are produced from O and the claim holds trivially. In the second case, p_i obtains a random permutation of those oracles from which it has received a REPORT-SIGS message. Then it repeatedly schedules an event *missingOutcome* after every $\Delta_{\text{req\_cert\_commit}}$. Whenever this event is triggered, p_i picks the next oracle out of the permutation and sends it a CERTIFIED-COMMIT-REQ message. These steps repeat until p_i obtains the outcome for *sn* in a CERTIFIED-COMMIT message.

In the worst case, i.e., when exactly the first f oracles of p_i's permutation are faulty, it takes at most $(f+1)\epsilon$ for processing the *missingOutcome* events plus a delay of $(f + 1)\Delta_{\text{req\_cert\_commit}}$ until the $(f + 1)$-st request is made. In addition, that last answer takes $2\Delta + \epsilon$ until it arrives back at p_i. Oracle p_i then consumes this and invokes *attestedReport* with another delay of at most ϵ. In summary, the time elapsed after τ may be up to $(f + 1)(\Delta_{\text{req\_cert\_commit}} + \epsilon) + 3(\Delta + \epsilon)$. □

Theorem 19. *Algorithm 7–8 implements a report attestation protocol.*

Proof. This follows directly from Lemmas 16–18. □

8.4 Transmission

There is one single instance of the transmission protocol that uses two events. An *attestedReport*(a) event may be invoked on the transmission protocol to *receive* an attested report a for transmission. The algorithm performs various checks and if these succeed, it *sends* a transaction containing a to C on the blockchain.

Internally the protocol accesses the functions *should-accept-finalized-report*(a) and *should-transmit-accepted-report*(a) of the reporting plugin with each attested report a. The protocol is defined with respect to synchronous time. More formally, we consider two events:

***attestedReport*(a):** Receives an attested report a for transmission.

***send transaction* with a:** Sends a blockchain transaction to C with a.

Every attested report $a = (sn, pos, \_\_, \_\_, \_\_)$ is identified by a sequence number *sn* of the corresponding outcome and the position *pos* within the vector of reports obtained from the outcome in the report attestation protocol. 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 report most recently committed by C on the blockchain, which is typically accessed by the *should-accept-attested-report*(a) and *should-transmit-accepted-report*(a) functions.

Definition 4. A *transmission* protocol satisfies for each correct oracle p_i:

Liveness: Let Δ_{transmit} be the transmission delay for an attested report a which some correct oracle p_i receives. If *should-accept-finalized-report*(a) returns TRUE and after time Δ_{transmit} *should-transmit-accepted-report*(a) also returns TRUE, then a blockchain transaction containing a is sent to C.

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

Theorem 20. *Algorithm 9 constitutes a transmission protocol.*

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

8.5 Summary

The Offchain Reporting Protocol (version 3.0) continuously collects observations made by the oracle nodes that run the protocol, converts the observations into reports, attests these reports, and finally transmits each report in one transaction to a smart contract running on a blockchain.

How do the formal properties of pacemaker, outcome generation, report attestation, and transmission according to Definitions 1–4 ensure this?

Let us first consider *liveness*. At a very high level and according to the timing model described in Section 3, the protocol is live only during "synchronous periods." Formally, this is captured by assuming there is initially *one* asynchronous phase, during which nodes and messages may be delayed arbitrarily and that lasts until GST. After GST, *one* synchronous phase follows, in which the network, the nodes, and their clocks behave timely.

In practice, *multiple* asynchronous and synchronous phases alternate. But since the combination of an initial, asynchronous phase followed by a synchronous phase can again be seen as a special case of a longer asynchronous phase, the model actually captures the situation of the system when it behaves again asynchronously later. The model implies in this way that the protocol is live during *every* sufficiently long period where the properties of the synchronous phase hold.

Liveness of OCR depends on the interaction between the pacemaker and outcome generation. Observe that the *eventual agreement* property of the pacemaker ensures that an epoch with a correct leader is started eventually, after GST. This epoch does not abort until the connected outcome generation indicates a leader change because the *liveness* property of outcome generation ensures that it outputs a *progress* event within each interval of length $\Delta_{\text{round}} + \Delta^+$, but the pacemaker would only abort the epoch if no such event occurs within $\Delta_{\text{progress}} > 2\Delta_{\text{round}} > \Delta_{\text{round}} + \Delta^+$.

Given that the leader is correct and that the epoch has sufficient length, the *liveness* property of the outcome generation protocol ensures that all correct oracles commit an outcome at least once within every interval of length Δ_{progress} because $\Delta_{\text{progress}} > 2\Delta_{\text{round}}$. This follows from the first condition of the *liveness* property in Definition 2. Hence, the correct oracles running outcome generation continuously commit outcomes.

The *timely inclusion* property of report attestation, in turn, ensures that for each such outcome, all reports are extracted, attested, and submitted to the transmission protocol with an *attestedReport* event. For an attested report a, the *liveness* property of the transmission protocol then applies: If the corresponding reporting-plugin functions determine that a should be transmitted, then a blockchain transaction containing a is sent to contract C. This completes the informal argument for why OCR is live.

Now we focus on the *safety* of the complete protocol. Recall that reporting starts with gathering observations, proceeds to reaching consensus on outcomes, then extracts reports, and finally transmits attested reports to a smart contract. The safety properties of the three main primitives guarantee that the transmitted reports accurately reflect the observations as follows.

The *integrity* property of outcome generation restricts all outcomes committed by the protocol to consist of the desired number of values observed by correct oracles. The same property also ensures that whenever the leader of the round, in which the observations are made, is correct, then the observations reflect the correct query, as dictated by the functions of the reporting plugin. A committed outcome therefore contains correctly made observations. According to the *integrity* property of report attestation, no report is output in an *attestedReport* event unless the report results from a committed outcome. Combined with the *safety* property of the transmission protocol, this reasoning shows that the protocol only transmits reports that result from correctly gathered observations. OCR therefore also respects safety.

Acknowledgments

We would like to thank Xiao Li, Georgios Tsimos, and Yann Vonlanthen for reviewing this document and helping refine it. We would also like to acknowledge Andrew Miller and Ari Juels for their work on earlier versions of this document.

References

[BCG20] M. Bravo, G. V. Chockler, and A. Gotsman. "Making Byzantine Consensus Live". In: *DISC*. Vol. 179. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 23:1–23:17.

[CGR11] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011.

[CL02] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461.

[DG04] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI*. USENIX Association, 2004, pp. 137–150.

[DLS88] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. "Consensus in the Presence of Partial Synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323.

[JNF20] M. M. Jalalzai, J. Niu, and C. Feng. "Fast-HotStuff: A Fast and Resilient HotStuff Protocol". In: *CoRR* abs/2010.11454 (2020). arXiv: 2010.11454. URL: https://arxiv.org/abs/2010.11454.

[NK24] O. Naor and I. Keidar. "Expected linear round synchronization: the missing link for linear Byzantine SMR". In: *Distributed Comput.* 37.1 (2024), pp. 19–33.

[PS17] R. Pass and E. Shi. "Hybrid Consensus: Efficient Consensus in the Permissionless Model". In: *DISC*. Vol. 91. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 39:1–39:16.

[Yin+19] M. Yin et al. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *PODC*. ACM, 2019, pp. 347–356.