LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs

LONGFEI QIU, Yale University, USA
YOONSEUNG KIM, Yale University, USA
JI-YONG SHIN, Northeastern University, USA
JIEUNG KIM, Inha University, South Korea
WOLF HONORÉ*, Yale University, USA
ZHONG SHAO, Yale University, USA

Byzantine fault-tolerant state machine replication (SMR) protocols, such as PBFT, HotStuff, and Jolteon, are essential for modern blockchain technologies. However, they are challenging to implement correctly because they have to deal with any unexpected message from Byzantine peers and ensure safety and liveness at all times. Many formal frameworks have been developed to verify the safety of SMR implementations, but there is still a gap in the verification of their liveness. Existing liveness proofs are either limited to the network level or do not cover popular partially synchronous protocols.

We introduce LiDO, a consensus model that enables the verification of both safety and liveness of implementations through refinement. We observe that current consensus models cannot handle liveness because they do not include a pacemaker state. We show that by adding a pacemaker state to the LiDO model, we can express the liveness properties of SMR protocols as a few safety properties that can be easily verified by refinement proofs. Based on our LiDO model, we provide mechanized safety and liveness proofs for both unpipelined and pipelined Jolteon in Coq. This is the first mechanized liveness proof for a byzantine consensus protocol with non-trivial optimizations such as pipelining.

CCS Concepts: • **Software and its engineering → Software verification**; **Distributed programming languages**; **Software safety**; **Formal software verification**; **Software reliability**; • **Theory of computation → Distributed computing models**; **Abstraction**.

Additional Key Words and Phrases: distributed systems, consensus protocols, byzantine fault-tolerance, safety, liveness, formal verification, refinement, proof assistants

ACM Reference Format:
Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. 2024. LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs. *Proc. ACM Program. Lang.* 8, PLDI, Article 193 (June 2024), 34 pages. https://doi.org/10.1145/3656423

1 INTRODUCTION

Byzantine State Machine Replication (SMR) protocols [Schneider 1990], such as PBFT [Castro 2001], HotStuff [Yin et al. 2019], and Jolteon [Gelashvili et al. 2022] form the basis of modern blockchain applications. They ensure that a linear history of a state machine is correctly replicated to a group

*Wolf Honoré is now at CertiK.

Authors' addresses: Longfei Qiu, Yale University, New Haven, USA, longfei.qiu@yale.edu; Yoonseung Kim, Yale University, New Haven, USA, yoonseung.kim@yale.edu; Ji-Yong Shin, Northeastern University, Boston, USA, j.shin@northeastern.edu; Jieung Kim, Inha University, Incheon, South Korea, jieungkim@inha.ac.kr; Wolf Honoré, Yale University, New Haven, USA, wolf.honore@yale.edu; Zhong Shao, Yale University, New Haven, USA, zhong.shao@yale.edu.

ACM 2475-1421/2024/6-ART193
https://doi.org/10.1145/3656423

of nodes and safe from tampering by a minority of malicious nodes. They are also called consensus protocols because a key part of these protocols is to make the participating nodes agree on a single history. As the open nature of public blockchains requires executing consensus protocols on a large number of nodes, in recent years, there has been a significant amount of research proposing new protocol designs that have better safety and liveness properties [Abspoel et al. 2021; Civit et al. 2022; Lewis-Pye 2022; Naor et al. 2021; Naor and Keidar 2020].

Despite the results that improve various aspects of byzantine SMR, it remains a herculean task to implement these protocols correctly, so that they enjoy the features claimed on paper. A paper description of a protocol is almost never sufficient to specify the behavior of a process under all possible situations. The unspecified aspects are open to interpretation, yet these details can have very subtle effects on the actual system. This issue is especially prominent under byzantine faults since the adversary now has more ways to influence the non-faulty nodes.

For a concrete example of this issue, we look at the pacemaker component of SMR protocols. Most SMR protocols are structured as an infinite sequence of smaller protocols called *rounds* or *views*, and each node participates in only one round at a time. The pacemaker drives the nodes to a new round when the current round is not making progress. As such, it plays a vital role in maintaining liveness.

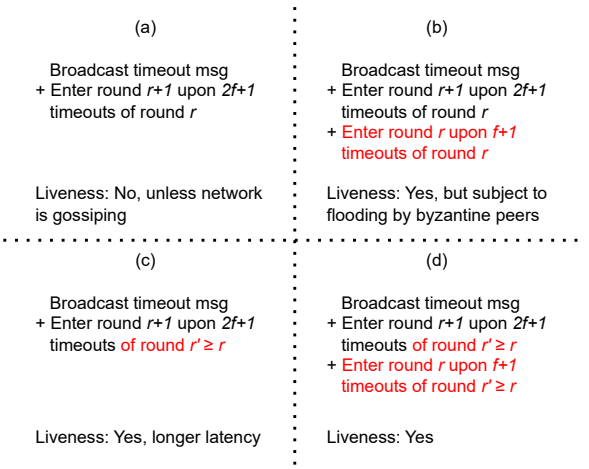

Fig. 1. Variants of a timeout-based pacemaker, with different liveness properties. Red text shows differences from (a). In variants (c) and (d), timeouts can come from different rounds.

The pacemaker usually consists of making each node broadcast a timeout message for its current round when no progress is observed within a certain period, and enter a new round after receiving a quorum of timeouts. In Fig. 1, we show four variants of this simple idea, with subtly different liveness properties. Notice that versions (b) and (d) differ only in allowing mixing timeouts from different rounds. This is significant because it allows non-faulty nodes to keep only the timeout of the highest round from each peer. Without this optimization, byzantine nodes can launch denial-of-service attacks by flooding non-faulty peers with timeouts, a tricky situation that would not occur under benign faults. This shows that paper proofs of protocols are not enough. We need proofs that can be directly tied to the implementation, which can only be achieved by machine-checked proofs on a formal model of the implementation.

Today, we have many formal frameworks for verifying the safety properties of distributed systems [Krogh-Jespersen et al. 2020; O'Hearn 2007; Sergey et al. 2017; Sharma et al. 2023]. In particular, formal safety proofs of consensus protocols have been studied in Carr et al. [2022]; Cirisci et al. [2023]; Honoré et al. [2021, 2022]; Rahli et al. [2018]; Taube et al. [2018]; Wilcox et al. [2015]. However, there are very few works that also establish formal liveness results for consensus protocols. IronFleet [Hawblitzel et al. 2015] and PSync [Drăgoi et al. 2016] establish liveness for benign-fault protocols such as Multi-Paxos [Lamport 1998], but do not handle byzantine faults. Padon et al. [2017] proposes a liveness-to-safety reduction approach for proving liveness, which has been applied to byzantine-fault protocols in Berkovits et al. [2019]; Losa and Dodds [2020], but their methodology has never been applied to partially synchronous protocols, which is the most common class of consensus protocols in practice.

Finally, none of the existing machine-checked liveness proofs are based on *refinement*, whereas safety results are usually stated as a refinement between the network model and the abstract interface. This situation is unsatisfactory for several reasons. Most importantly, it obscures high-level reasoning and prevents proof reuse since every definition and every lemma is tied to network-level details. It also poses a challenge to users of the system since they must understand the implementation details of the system in order to understand what is proved as "liveness."

In this work, we aim to simplify the task of constructing liveness proofs of byzantine consensus protocols. We achieve this by introducing an intermediate model of consensus between the SMR interface and network details that supports proving both safety and liveness via refinement. Our key insight is that existing models lack a representation of the pacemaker, which, as we have seen, is critical to liveness, and this prevents them from handling liveness. We start from the Atomic Distributed Object (ADO) model, which has been used to verify the safety of several benign-fault protocols [Honoré et al. 2021, 2022]. We show that by adding the pacemaker state into the model, the liveness of consensus can be reduced to a few safety properties on timed traces, which can be easily proved through refinement. We also introduce *segmented traces*, a variant of timed traces that enables more effective formalization of liveness properties and proofs. Using our LiDO model, we obtain safety and liveness proofs for both unpipelined and pipelined Jolteon [Gelashvili et al. 2022].

To summarize, our contributions are:

- **LiDO**, a model of consensus formalized in Coq, supporting both safety and liveness reasoning via refinement;
- **Segmented traces**, an effective formalism for proving liveness properties via refinement;
- **Implementations of both unpipelined and pipelined Jolteon in Coq**, providing case studies for our methodology;
- **Refinement-based proofs** of both safety and liveness of Jolteon using our LiDO model.

All proofs have been mechanized in Coq and are available as artifacts [Qiu et al. 2024]. We have also extracted unpipelined Jolteon into an OCaml executable, showing that our network model is reasonable and realistic.

2 OVERVIEW

2.1 Background: State Machine Replication Under Partial Synchrony

The Partial Synchrony Assumption. Message-passing distributed systems rely on getting messages delivered to make progress. Therefore any liveness property of such systems depends on assumptions about message delivery. Depending on the kind of assumptions they make, the systems are classified into *asynchronous*, *synchronous*, or *partially synchronous* protocols [Dwork et al. 1988]. This work targets proving the safety and liveness of partially synchronous SMR protocols. Our theory can be applied to both benign-fault and byzantine-fault tolerant protocols, but in this work, we mainly consider Byzantine fault-tolerant (BFT) protocols.

There are two versions of partial synchrony [Dwork et al. 1988]. In one version, there is a fixed upper bound Δ of message delivery latency, but it only holds after a certain timepoint called *global synchronization time* (GST). The participating processes know Δ but do not know when GST commences. In another version, the delivery latency is always bounded, but the processes do not know the exact bound. In this work, we use the first version, as it is easier to work with formally.

State Machine Replication. The safety definition of SMR is well-known. Clients submit *requests* to the system. The system outputs *responses* to requests, each request is responded to at most once, and the request-response trace must linearize to an atomic spec of a state machine [Herlihy and Wing 1990]. Under partial synchrony, the system processes do not know when GST begins, so they

cannot rely on messages being delivered in time. As such, they maintain safety even during periods of asynchrony.

The liveness definition of SMR is more subtle. There can be two reasonable definitions:

Definition 2.1. An SMR system is live if every client request is eventually responded to.

Definition 2.2. An SMR system is live if it responds to client requests infinitely often.

There is a gap between the two definitions. Under Definition 2.2, the system may selectively respond to a subset of requests. When there is only a fixed number of clients, we can simply make the system choose among client requests in a round-robin fashion, so that each client is fairly serviced. When the SMR clients are unbounded in number, as in public blockchains, maintaining fairness among clients is a research problem that is beyond the scope of this work [Kelkar et al. 2020; Kursawe 2020]. Therefore, this work aims to establish Definition 2.2.

2.2 The ADO Model of Consensus

Although the safety and liveness definitions of SMR are simple and intuitive, proving that an implementation satisfies these definitions is not. As one of the early attempts, Verdi [Wilcox et al. 2015; Woos et al. 2016] used 50,000 lines of code to prove the safety of Raft but did not verify liveness. Proofs of this complexity are difficult to maintain and difficult to port to other implementations.

To better manage the complexity of proofs, a successful strategy is to introduce an intermediate abstraction between SMR and the network model. The abstraction captures essential information about the network state but remains simple enough to allow easy reasoning about system behavior. Most notably, the Atomic Distributed Object (ADO) theory has been proposed to verify the safety of multiple benign-fault consensus protocols, including Raft with reconfiguration [Honoré et al. 2021, 2022]. Here we give an intuitive introduction to ADO. The formal details are in Section 3.

The ADO theory gives a detailed view of how the consensus log grows during the execution of a consensus protocol. It models the execution of a protocol as a group of proposer processes interacting with a concurrent object. The basic idea of ADO is that within each round of consensus, three events occur in sequence: first, the leader is given an up-to-date branch of the consensus log; then, it appends one or more requests at the tip of the branch; finally, it attempts to commit its changes. These steps are called *pull*, *invoke*, and *push*. In each step the leader may collect enough votes and succeed, or it may fail to collect votes before the pacemaker drives it to the next round.

To model this behavior, the ADO object exposes three operations $Pull(r)$, $Invoke(r, m)$, and $Push(r)$, where r is the round the proposer participates in; m is the request (called *method* in ADO theory) the proposer wishes to append. The object responds to each call with either *Success* or *Timeout*. When it responds to a call, a *cache node* is created to record information about the response. Successful responses to *Pull*, *Invoke*, *Push* correspond to *ECache*, *MCache*, *CCache* respectively, where E, M, C stand for Election, Method invocation, and Commit. The cache nodes are chained together by causal relation to form a *cache tree*.

Fig. 2 shows an example cache tree. Each *MCache* represents a client method that has been proposed by a proposer. We say an *MCache* is *committed* if there exists a path from that *MCache* to a *CCache*. The safety property of an ADO object is an invariant of the cache tree: there exists a path from *Root* that contains all committed *MCache*. It follows that we can take the sequence of all committed *MCache* on this path as the consensus log, and implement SMR on top of it, as was done in Honoré et al. [2021, 2022]. The linearization point of each client method is the point where the corresponding *MCache* becomes committed. Hence, SMR liveness is equivalent to creating *MCache* and *CCache* infinitely often in the ADO cache tree.

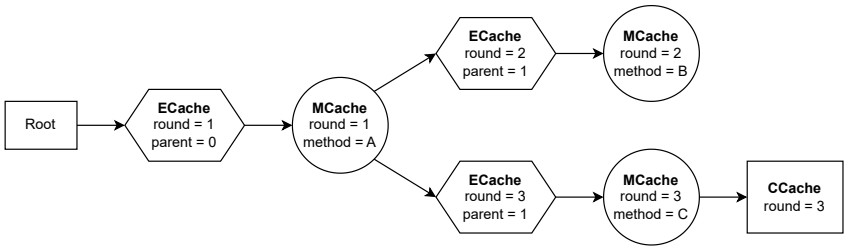

Fig. 2. An Example ADO Cache Tree. An *MCache* is committed if there exists a path from it to a *CCache*. Hence *MCache* of round 1 and 3 are committed, but *MCache* of round 2 is not.

2.3 The Need for a New Model

The ADO model nicely abstracts out the common logic of safety proofs, but it has not been useful for verifying liveness. We now look at why proving liveness remains difficult. Intuitively, a refinement-based liveness proof should involve the following steps:

(1) Among the valid traces of the abstract model, we identify a subset of *live* traces;
(2) We prove that all live traces of the abstract model satisfy SMR liveness (Definition 2.2);
(3) We identify the live traces of the implementation;
(4) We prove that every live trace of the implementation refines a live trace of the model.

Clearly the key part of this plan is the first step. We need to carefully define the model and its live traces so that it is both easy to prove that every live trace satisfies SMR liveness, and to prove that an implementation refines the live traces.

Temporal properties are easiest to work with when they are posed as *safety* properties. That is, they concern system dynamics over only a finite period of time. For example, "the system commits client methods infinitely often" is a liveness property, but "the system will commit one client method within each period of 10Δ" is a safety property. Ideally, when we define live traces of the abstract model and the network model, we should always characterize them using safety properties.

We now try to execute the plan on the ADO model. As discussed above we have to create a *CCache* infinitely often. To reduce this to a safety property, naively, we may try:

Example 2.3. After GST, when a non-faulty proposer calls $Push(r)$, it receives *Success* within 2Δ.

If we could prove this, and we arrange non-faulty proposers to call *Push* infinitely often, we could show that a *CCache* is created infinitely often. At first glance, this seems intuitive. At the network level, a call from a non-faulty proposer usually corresponds to it broadcasting a request message. Since the message will be delivered to every non-faulty voter within Δ, and the votes will come back within Δ, the request will succeed within 2Δ.

However, in making this inference we have neglected an important factor: the pacemaker. By influencing the round-change process, the adversary may obstruct liveness in a number of ways:

- The non-faulty nodes may never enter round r, so the leader may never make its request;
- The byzantine nodes may initiate a round-change before the request succeeds, so the leader will not receive a *Success* response.

Even if byzantine nodes do nothing, the non-faulty nodes will still initiate round-change when their timers expire. Therefore to prove that a request will succeed, we at least need to assume that all non-faulty nodes still have sufficient time in their timers. Unfortunately, the ADO model does not capture information about these local timers, preventing us from formally expressing this assumption. This clearly shows that we need a new model that incorporates the pacemaker information we need.

This motivates us to propose the LiDO model. As shown in Fig. 3, we add two state variables *round* and *remaining time* (*rem_time* for short). These variables represent a logical timer: *round* represents the round the voters currently guarantee liveness for, while *rem_time* is the least amount of time the voters promise not to timeout. In this work, we will consistently use Δ as a unit of time, so $rem\_time = 3$ intuitively means none of the non-faulty voters will timeout within 3Δ.

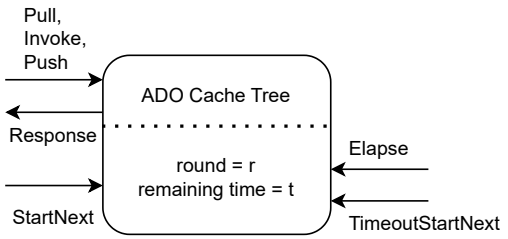

Fig. 3. The LiDO object architecture.

The timer variables can only be manipulated through a number of calls, shown in Fig. 3. In particular, $Elapse()$ represents the flow of time: it decreases *rem_time* by 1. The call $TimeoutStartNext()$ increases *round* by 1, but it can only be called when $rem\_time = 0$. This says that the adversary cannot terminate a round until the logical timer has expired. The adversary also cannot make time flow too fast. We capture this by allowing $Elapse()$ to be called at most once per period of Δ. Hence if $rem\_time = 3$, then the adversary must call $Elapse()$ three times, taking a period of 3Δ, before it may increase *round*. Our safety rules on $Elapse()$ and $TimeoutStartNext()$ thus formalize the notion that the adversary cannot preempt a non-faulty leader too soon. We also allow the leader of round r to call $StartNext(r)$, a request to start round $r + 1$, after all requests in round r have succeeded. More formal details are in Section 3.2.

With pacemaker information added to the model, we can weaken Example 2.3 to

Example 2.4. After GST, if $round = r$ and $rem\_time \geq 2$, the leader of round r is non-faulty and calls $Push(r)$, it receives *Success* within 2Δ.

This is now intuitively implementable because the pacemaker will not intervene within 2Δ.

2.4 Proving Liveness Under Partial Synchrony

In the previous sections we gave an intuitive explanation of our LiDO model design but did not give any formal liveness details. We now introduce our liveness formalism.

In synchronous or partially synchronous systems, the most general formalism for characterizing live traces is timed traces [Lamport 2005]. We first assume there exists a *time* variable that represents physical time and increases continuously. When each event e occurs, we pair it with the current reading of *time*. The trace thus consists of a sequence of pairs $(e_0, time_0); (e_1, time_1); \cdots$ with $time_0 \leq time_1 \leq \cdots$. The problem is that continuous time is difficult to encode in proof checkers.

We can look at timed traces in a different way. In general, we only consider *non-Zeno* timed traces, meaning only a finite number of events may occur within a finite period of time. Hence assume the set of all events occurred before any timepoint t is always finite. Then we can cut the trace into segments, each representing a period of Δ, and use them to cover the entire trace. Formally:

Definition 2.5. A *segmented trace* is an arbitrary-length sequence of finite untimed traces $(\tau_0, \tau_1, \cdots)$, such that each τ_i is a valid trace, and each τ_i is a prefix of τ_{i+1}.

Definition 2.6. Let T be any timepoint with $T \geq GST$. We define the (T, Δ)-segmentation of a non-Zeno timed trace τ to be $(\tau_0, \tau_1, \cdots)$ where τ_i is the sequence of all events that occurred at some timepoint $t < T + i\Delta$. If τ is infinite, the segmentation is also infinite; if τ is finite and only covers events up to timepoint $T + k\Delta$, the segmentation also ends at τ_k.

In Definition 2.6, the requirement that $T \geq GST$ ensures that all events occurred before GST are hidden into τ_0, so we do not need to worry about whether each segment between τ_i and τ_{i+1} is before or after GST.

Segmented traces provide a convenient formalism for stating temporal properties with time constraints. To see this, we look at formalizing the partial synchrony assumption in a network model. In this work, the formal definition of partial synchrony under timed traces is:

Assumption 2.7. *If process p sends msg to process q at timepoint t, both p, q are non-faulty, then process q receives msg at least once in the interval $[t, \max\{t, GST\} + \Delta]$.*

For a valid trace τ, let $msg(\tau)$ be the set of all messages already sent within τ, and let $deliv\_msg(\tau)$ be the set of delivered messages, represented as (id, msg) pairs. For each message m, let $sender(m)$ be its sender and $recip(m)$ its recipient set. Let H be the set of non-faulty processes. Then we have:

Lemma 2.8. *In every (T, Δ)-segmentation of every live timed trace of a network model, we have*

$$\forall i, \forall m, \forall p, m \in msg(\tau_i) \Rightarrow sender(m) \in H \Rightarrow p \in recip(m) \Rightarrow p \in H \Rightarrow (p, m) \in deliv\_msg(\tau_{i+1}).$$

Proof: If m was sent before GST then it is delivered at least once before $GST + \Delta$. If it was sent within the interval $[GST, GST + i\Delta)$ then it is delivered at least once before $GST + (i+1)\Delta$.

Thus we can take Lemma 2.8 as the definition of partial synchrony under segmented traces.

Refinement of Segmented Traces. We now observe that segmented traces enjoy a natural notion of refinement. Let f be a refinement mapping between a spec system and an implementation system. Then f maps traces of the implementation to traces of the spec. Furthermore, the definition of a refinement mapping requires that, if τ is a prefix of τ', then $f(\tau)$ is also a prefix of $f(\tau')$.

It follows, if $(\tau_0, \tau_1, \cdots)$ is a valid segmented trace of the implementation, then $(f(\tau_0), f(\tau_1), \cdots)$ is also a valid segmented trace of the spec. We say that $(\tau_0, \tau_1, \cdots)$ refines $(f(\tau_0), f(\tau_1), \cdots)$.

Thus, segmented traces are a very convenient formalism for analyzing partially synchronous systems. They are easy to encode in proof checkers, and it is easy to define refinement between them. Throughout this work, we will use segmented traces as the main formalism for analyzing liveness. Our plan consists of the following steps:

(1) We specify the LiDO model of consensus as our spec (Section 3);
(2) We specify a set S of segmented traces as the live traces of the LiDO model, and prove that they satisfy SMR liveness (Section 3.3);
(3) We specify a system model for implementing unpipelined and pipelined Jolteon (Section 4.1);
(4) We define a set S' of segmented traces as live traces of the implementation, and show that S' covers all timed traces of the implementation (Definition 4.3);
(5) We establish refinement mapping between the implementations and LiDO, and prove that every trace in S' refines a trace in S (Section 4.3).

Layered Refinement Proof. Fig. 4 shows a schematic diagram of our refinement proof between LiDO and Jolteon. In between LiDO and the network model, we used two additional layers called the Server and Voting layer. These layers implement the LiDO model in a shared-memory manner. This allows us to focus on the important invariants maintained by the protocol while ignoring details such as message delivery and bookkeeping, which only appear in the network model. Introducing these intermediate layers simplifies overall engineering effort. See Section 4 for details.

3 THE LIDO MODEL OF CONSENSUS

In this section, we formally define the LiDO model and its live traces. We first define the ADO model as a concurrent object, and then define the LiDO model as an extension of ADO.

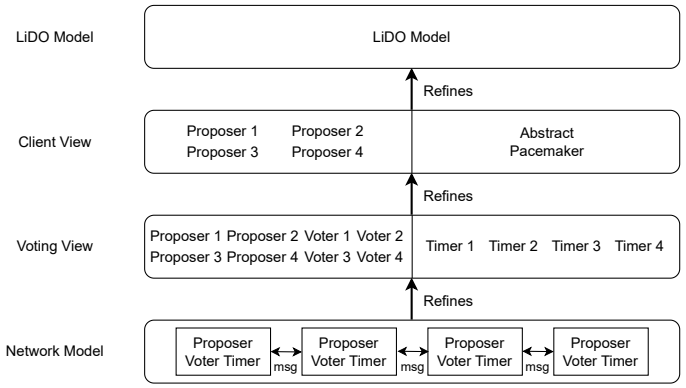

Fig. 4. Jolteon safety refinement proof architecture.

3.1 The ADO Model

Algorithm 1 The Method Pool Object

```
1: initialize: Q ← ∅
2: upon RegisterMethod(m):
3:     Q ← Q ∪ {m}
4: upon CheckMethod(m):
5:     return m ∈ Q
```

In byzantine consensus, although byzantine proposers can send any message, the system is required to maintain *external validity*, meaning all requests committed in the log must come from external clients, not fabricated by byzantine proposers. Therefore, we first assume there exists an object called the *method pool* (Algorithm 1). The object state is a set Q of client-signed methods. Initially, $Q = \emptyset$. The object exposes two operations $RegisterMethod(m)$ and $CheckMethod(m)$. SMR clients may call $RegisterMethod(m)$ to add a method m into Q. The ADO object may call $CheckMethod(m)$ to check whether m has been registered or not. Both calls are atomic. In an implementation, they correspond to the client signing a request, and the voters checking the signature.

The ADO object proper is a concurrent object, formalized as a transition system consisting of two kinds of events: an agent making a call on the object, and the object responding to a call. The object does not need to respond to each call immediately; it may respond to it at some arbitrary later time, but no changes to the object state occur before the response. We assume each agent is sequential: it only waits upon one call at a time. Thus in a valid trace, each agent alternates between making a call and receiving its response, and these events can be interleaved.

The agents interacting with the ADO object are the proposers of a consensus protocol. In the standard setting, there are $3f + 1$ proposers, of which $2f + 1$ are non-faulty proposers and f are byzantine. We assume that in the consensus protocol, rounds are numbered from 1, and in each round, one of the proposers is predetermined as the leader. We use $leader\_at(r)$ to represent that leader. Assume that each proposer becomes a leader infinitely often.

The object state of ADO is a set Σ of cache nodes that form a cache tree. Therefore, we first formally define cache nodes and the cache tree, then define the ADO object. The cache node structure is defined in Fig. 5 (a). Each cache node except *Root* contains a *round* field, along with other data. Let Σ be a set of cache nodes with at most one *ECache*, one *MCache*, and one *CCache* per round. We use $\Sigma[r].ecache, \Sigma[r].mcache, \Sigma[r].ccache$ to represent that unique cache node of round r. We write $\Sigma[r].ecache = \bot$ if round r does not have an *ECache*, similarly for other notations.

For each *ECache*, *MCache*, *CCache* in Σ, we define its *parent* as in Fig. 5 (b). The *cache tree* of Σ is a graph with all cache nodes except *TCache* as its vertices, and directed edges from each node to its direct children as its edges. The cache tree is well-defined when the parent of each node is

$$
\begin{aligned}
CacheNode \triangleq\ & Root \\
& \mid ECache(\mathbb{N}_{round} * \mathbb{N}_{parent\_round}) \\
& \mid MCache(\mathbb{N}_{round} * Method) \\
& \mid CCache(\mathbb{N}_{round}) \\
& \mid TCache(\mathbb{N}_{round})
\end{aligned}
$$

(a) Cache Nodes

$$
\begin{aligned}
parent(ECache(r,p)) &\equiv \begin{cases} Root & (p = 0) \\ \Sigma[p].mcache & (p > 0) \end{cases} \\
parent(MCache(r,m)) &\equiv \Sigma[r].ecache \\
parent(CCache(r)) &\equiv \Sigma[r].mcache
\end{aligned}
$$

(b) Cache Node Parent Relation

Fig. 5. Definition of ADO Cache Nodes and Node Parents.

$$
\begin{aligned}
PullPre(\Sigma, r, p) &\equiv p < r \wedge (p = 0 \vee \Sigma[p].mcache \neq \bot) \wedge \Sigma[r].ecache = \bot \\
&\quad \wedge (\forall r', r' < r \Rightarrow (p \geq r' \vee \Sigma[r'].ccache = \bot)) \\
InvokePre(\Sigma, r) &\equiv \Sigma[r].ecache \neq \bot \wedge \Sigma[r].mcache = \bot \\
PushPre(\Sigma, r) &\equiv \Sigma[r].mcache \neq \bot \wedge (\forall r', r' > r \Rightarrow (\Sigma[r'].ecache = \bot \vee \Sigma[r'].ecache.parent\_round \geq r))
\end{aligned}
$$

Fig. 6. Preconditions for *Success* response of ADO object. For explanations, see Appendix A.

well-defined, and the graph forms a rooted tree with *Root* as its root. See Fig. 2 for an example cache tree. We say an *MCache* is *committed*, if there exists a path in the cache tree from that *MCache* to a *CCache*. Hence in Fig. 2, $\Sigma[1].mcache$ is committed, as there is a path to $\Sigma[3].ccache$, but $\Sigma[2].mcache$ is not.

When the cache tree is well-defined, there is a unique path from *Root* to each cache node c. The sequence of all *MCache* on that path forms the *consensus log* up to node c. We denote it by $log(c)$.

We now define the ADO object. The object state is a set Σ of cache nodes. Initially, $\Sigma = \{Root\}$. The object exposes three operations, which are $Pull(r)$, $Invoke(r, m)$, and $Push(r)$ with $r > 0$. Only $leader\_at(r)$ may call $Pull(r)$, $Invoke(r, m)$, or $Push(r)$. Also, $Invoke(r, m)$ can only be called when m has been registered in the method pool. Otherwise, it is considered an invalid call and fails immediately with no change in object state.

The object may respond to each call with either *Success* or $Timeout$. If the caller is byzantine, it may voluntarily stop waiting for its current call and we represent this with a special response *Dropped*, with no change in object state.

The object may always respond to a call with $Timeout$, adding a $TCache$ to Σ. When it decides to respond to $Pull(r)$ with *Success*, it must non-deterministically choose a p such that $PullPre(\Sigma, r, p)$ is currently satisfied. Similarly, when it responds to $Invoke(r, m)$ or $Push(r)$ with *Success*, the conditions $InvokePre(\Sigma, r)$ or $PushPre(\Sigma, r)$ must be currently satisfied, respectively. The definitions of these conditions are shown in Fig. 6. The changes to the object state upon each response are defined in Algorithm 2.

Linearizability of ADO. In Honoré et al. [2021, 2022], the ADO object was described as an atomic object. The refinement proof works by reordering network events to their linearization points. However, for liveness refinement, we have to define a refinement mapping between ADO and the network model, and events cannot be reordered, which forces us to switch to a concurrent spec.

Nevertheless, we can define an atomic version of ADO as follows. The object exposes exactly the same interface as concurrent ADO. However, when any proposer makes a call, the object atomically chooses a response (*Success* or $Timeout$) and returns it immediately. The preconditions and effects of each response are exactly the same as defined in Fig. 6 and Algorithm 2.

Lemma 3.1. *The concurrent ADO object is linearizable to the atomic ADO object.*

Algorithm 2 ADO Object State Changes

initialize: $\Sigma \leftarrow \{Root\}$
upon return *Success* to call $Pull(r)$:
    Choose p such that $PullPre(\Sigma, r, p)$ is satisfied, if no such p exists return $Timeout$ instead.
    $\Sigma \leftarrow \Sigma \cup \{ECache(r, p)\}$
upon return *Success* to call $Invoke(r, m)$:
    Check $InvokePre(\Sigma, r)$ is satisfied, if not return $Timeout$ instead.
    $\Sigma \leftarrow \Sigma \cup \{MCache(r, m)\}$
upon return *Success* to call $Push(r)$:
    Check $PushPre(\Sigma, r)$ is satisfied, if not return $Timeout$ instead.
    $\Sigma \leftarrow \Sigma \cup \{CCache(r)\}$
upon return $Timeout$ to call $Pull(r)$, $Invoke(r, m)$, or $Push(r)$:
    $\Sigma \leftarrow \Sigma \cup \{TCache(r)\}$

Proof: We simply choose the point where the object generates a response as the linearization point of a call. The call does not have any effect on the object state before the response is generated. The preconditions of generating a response depend only on the object state at the response point. Therefore, moving every call event to the response point results in a valid atomic trace with the same final object state.

Safety of ADO. In Appendix A, we give a presentation of the safety theory of ADO. Here we simply understand that, the ADO cache tree is always well-defined, and there is always a path starting from *Root* that contains all committed *MCache*. Let c be the committed *MCache* with the highest round number, then we can take $log(c)$ to be the current committed consensus log.

Implementing ADO. We also define what it means that a network system with byzantine processes implements the ADO object. Let M be the message space of the system, the set of all possible messages that may be created within the system. For every reachable system state z, let $msg(z) \subseteq M$ denote the set of all messages that have been actually created at state z. Then we define:

Definition 3.2. A refinement between the ADO object and a network system consists of the following data:

(1) A refinement mapping f that maps valid finite network traces to valid finite traces of ADO object, which defines correspondence between network state z and ADO cache tree Σ;
(2) For each possible cache node c, a certificate set $cert(c) \subseteq M$, such that in every corresponding pair of network state z and ADO cache tree Σ, we have $c \in \Sigma$ iff at least one member of $cert(c)$ is in $msg(z)$.

Although byzantine processes can send any message, they still have to follow cryptographic restrictions, which is why they cannot fabricate messages in $cert(c)$ to claim the existence of a cache node. Thus whatever byzantine processes do in the network system, the net effect is still *as if* they are following the ADO interface. Hence external processes such as SMR clients and executors can use *CCache* certificate messages as evidence that a method is committed, and act accordingly.

3.2 The LiDO Model

We now define the LiDO model, which is the ADO model extended with state variables and operations that represent an abstract pacemaker.

As shown in Fig. 3, the LiDO object state consists of an ADO cache tree Σ, and two integers *round* and *rem_time*. The object exposes six operations. The $Pull(r)$, $Invoke(r, m)$, and $Push(r)$ operations affect the cache tree, and their semantics are exactly the same as the ADO object (Algorithm 2). There are three new operations $StartNext(r)$, $Elapse()$, and $TimeoutStartNext()$ which affect the pacemaker state and are described below.

We introduce a new agent called the adversary $\mathcal{A}$, which represents the effect of time flowing. $\mathcal{A}$ may call $Elapse()$, which decreases *rem_time* by 1. When $rem\_time = 0$, $\mathcal{A}$ may call $TimeoutStartNext()$ to increase *round* by 1 and reset *rem_time* to a preconfigured value *reset_val*. This models a logical timer that is simulated by the local timers of each voter. It allocates a fixed duration for each round, and when the timer expires, the pacemaker may intervene to start the next round. Both $Elapse()$ and $TimeoutStartNext()$ are atomic calls: the object responds to the call immediately.

Algorithm 3 Abstract Pacemaker State Changes

```
1:  reset_val: Implementation-defined constant for rem_time reset value.
2:  initialize:
3:      round ← 1
4:      rem_time ← reset_val
5:  upon respond to StartNext(r):
6:      if round = r then
7:          round ← r + 1
8:          rem_time ← reset_val
9:  upon Elapse():
10:     if rem_time > 0 then
11:         rem_time ← rem_time − 1
12: upon TimeoutStartNext():
13:     if rem_time = 0 then
14:         round ← round + 1
15:         rem_time ← reset_val
```

We allow $leader\_at(r)$ to call $StartNext(r)$. This call sends a signal to the pacemaker that it may start round $r + 1$ without waiting for the timer of round r to expire. This is a concurrent call: the object does not need to respond to the signal immediately.

The formal effects of these calls are shown in Algorithm 3.

3.3 The Live Traces of LiDO

We now define the live traces of LiDO. In general, we define live traces by *liveness requirements*. A valid segmented trace $(\tau_0, \tau_1, \cdots)$ is a live trace whenever it satisfies these requirements. Our requirements only concern events over a fixed-length duration. This makes them safety properties which are easy to handle using refinement.

The liveness requirements on LiDO are divided into protocol-independent ones and protocol-dependent ones. The reason is that pipelined protocols provide a weaker liveness guarantee, as it needs the cooperation of two (or more) leaders to commit a method, so certain liveness properties of unpipelined protocols are not enjoyed by pipelined ones. Here we focus on unpipelined protocols. The liveness of pipelined protocols will be discussed in Section 5.

Definition 3.3. The protocol-independent liveness requirements are:

(1) Between τ_i and τ_{i+1}, $Elapse()$ is called at most once;
(2) If $rem\_time(\tau_i) > 0$, between τ_i and τ_{i+1} if $Elapse()$ is not called then *round* is increased at least once;
(3) There exists constant C, such that if $rem\_time(\tau_i) = 0$, then $round(\tau_{i+C}) > round(\tau_i)$;

By "between τ_i and τ_{i+1}," we mean the trace τ_{i+1} with prefix τ_i removed. Together, these requirements ensure that the round number will increase unboundedly in an infinite trace, while still giving sufficient time to each round.

Many protocols allow two consecutive non-faulty leaders to cooperate to start a new round, without waiting for the timer to expire. This feature can be formulated as an additional liveness requirement:

Definition 3.4. If $round(\tau_i) = r$, both $leader\_at(r), leader\_at(r+1)$ are non-faulty, $leader\_at(r)$ has called $StartNext(r)$ in τ_i, then $round(\tau_{i+1}) \geq r+1$.

Definition 3.5. The protocol-dependent liveness requirements for unpipelined protocols are:

(1) A non-faulty leader calls $StartNext(r)$ only after a *CCache* in round r is created;
(2) There exists constant $N < reset\_val$, such that if $round(\tau_i) = round(\tau_{i+N}) = r, rem\_time(\tau_i) \geq N, leader\_at(r)$ is non-faulty, then a *CCache* of round r is created by the end of τ_{i+N}.

The second requirement simply says that, given round change does not occur, each non-faulty leader will always commit a method by itself. Recall that there are only two ways to increase *round*: either the leader calls $StartNext(r)$ or the adversary calls $TimeoutStartNext()$. When a new round r starts, rem_time is reset to $reset\_val > N$. The adversary cannot call $TimeoutStartNext()$ within $N\Delta$. Hence if *round* is increased within $N\Delta$, then $leader\_at(r)$ must have called $StartNext(r)$, which implies a *CCache* of round r has been created, by the first rule. If *round* is not increased, then a *CCache* must have been created as well, by the second rule. Therefore these rules guarantee that as soon as *round* reaches r, the leader of round r will commit a method within $N\Delta$.

We observe that the liveness proofs of many byzantine consensus protocols, including PBFT [Castro 2001], HotStuff [Yin et al. 2019], and Jolteon [Gelashvili et al. 2022] follow the same reasoning pattern as outlined above. Their differences mainly lie in (1) the round-change mechanism implementation and (2) how each leader utilizes its allocated time.

As a simple example, we may assume that after *round* is increased to r, the leader of round r will learn this fact within Δ. Upon learning this fact, it immediately makes *Pull*, *Invoke*, and *Push* calls in sequence, and each call takes at most 2Δ to succeed. In Section 4, we will see that unpipelined Jolteon follows exactly this pattern, except that it performs *Pull* and *Invoke* simultaneously in one phase. We can formalize the above assumptions as follows:

(1) If $round(\tau_i) = r$, $rem\_time \geq 1$, $leader\_at(r)$ is non-faulty, but no *ECache* of round r has been created, then either $leader\_at(r)$ is currently waiting upon $Pull(r)$, or it will call $Pull(r)$ between τ_i and τ_{i+1};
(2) If $round(\tau_i) = r$, $leader\_at(r)$ is non-faulty, $rem\_time \geq 2$, and the leader is waiting upon a call $Pull(r), Invoke(r, m), Push(r)$, then that call will succeed before the end of τ_{i+2};
(3) When a non-faulty leader receives *Success* for $Pull(r)$, it immediately chooses m and calls $Invoke(r, m)$; similarly it immediately calls $Push(r)$ after $Invoke(r, m)$ succeeds.

By "immediately," we mean between τ_i and τ_{i+1}, if the first event has occurred, then the second must have also occurred. Since the leader takes Δ to enter round r, and each phase takes 2Δ, the leader will commit a method within 7Δ. Hence take $reset\_val = 8$ and Definition 3.5 is satisfied.

Liveness of LiDO. We now formally state and prove LiDO's liveness property (for unpipelined protocols), which implies SMR liveness.

Theorem 3.6. *In every infinite live trace $(\tau_0, \tau_1, \cdots)$ of LiDO, let $r = round(\tau_0)$, then for every $r' > r$ with $leader\_at(r')$ non-faulty, there exists i such that a CCache of round r' is created by the end of τ_i.*

Proof: Let $(\tau_0, \tau_1, \cdots)$ be an infinite live trace. We first show that $round(\tau_i)$ increases unboundedly. Let $r = round(\tau_i), t = rem\_time(\tau_i)$. If $round(\tau_i) = round(\tau_{i+1})$ but $t > 0$ then $Elapse()$ is called once between τ_i and τ_{i+1}, which decreases rem_time by 1. Hence if $round(\tau_i) = round(\tau_{i+t})$, then $rem\_time(\tau_{i+t}) = 0$. By assumption, there exists constant C such that $round(\tau_{i+t+C}) > round(\tau_i)$.

Since *round* increases unboundedly, for each $r > round(\tau_0)$, we can find i such that $round(\tau_i) < r$ but $round(\tau_{i+1}) \geq r$. By assumption, there exists constant N such that if $\tau_{i+1} = \tau_{i+1+N} = r$, then

there is *CCache* of round r by the end of τ_{i+1+N}. On the other hand, if $\tau_{i+1+N} > r$, then $leader\_at(r)$ must have called $StartNext(r)$, so there exists a *CCache* of round r by the end of τ_{i+1+N} as well.

Given that each non-faulty proposer becomes a leader infinitely often, this implies that *CCache* is created infinitely often during execution. Since each *CCache* represents a new method being committed, we see that new methods are committed infinitely often.

4 PROVING SAFETY AND LIVENESS OF UNPIPELINED JOLTEON

In Section 3 we fully defined the LiDO object and its live traces. In this section we use unpipelined Jolteon as an example to show how to prove a network model refines LiDO. We study Jolteon because its design maps nicely onto the ADO's three-step view of consensus.

4.1 System Model

We consider a system consisting of a fixed finite set of non-faulty and byzantine processes. The only way of communication between these processes is through sending and delivering messages. The set M of all messages that may potentially be created within the system is called its *message space*. The set Z of all internal states each non-faulty process may potentially reach during execution is called its *state space*. Within set Z, a special state z_0 is designated as the initial state of each non-faulty process. We do not model the internal state of byzantine processes.

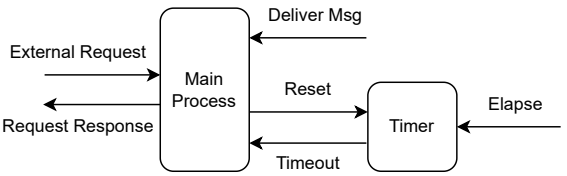

Fig. 7. Architecture of each non-faulty process.

The system state consists of three parts: 1) A finite map $proc\_state$ from process IDs to the current state of that process; 2) A finite set msg of all messages that have been created within the system; 3) A finite set $deliv\_msg$ of process-message pairs, indicating which messages have been delivered to which processes. Initially, $proc\_state(pid) = z_0$ for each process, and $msg = deliv\_msg = \emptyset$.

Fig. 7 shows the architecture of each non-faulty process. It is specified as a main process with a timer object attached. The main process has three operations: it can receive requests from external clients, receive messages from the network, and receive timeout signals. Only the timer can send timeout signals to the main process. The timer object has two operations called *reset* and *elapse*, where *reset* can only be called by the main process while *elapse* is an external signal. The formal details of the timer are explained later.

Each event that may occur within the system belongs to one of the following kinds:

(1) Deliver an external client request to a non-faulty process;
(2) Deliver a message to a non-faulty process, provided it has been sent previously;
(3) Deliver a time-elapse signal to a timer object;
(4) A byzantine process sends an arbitrary message, subject to constraints (explained later).

The action of a non-faulty process upon each delivery event is specified by a handler function. The action may involve state changes and sending messages and is atomic with the event.

The Timer Object. We now study the timer object more closely. Normally, a timer is considered a continuous object that exposes a single operation *reset* and sends out timeout signals. After GST, the timer sends out a timeout signal when and only when a predetermined duration δ has elapsed from the most recent reset call. This model is intuitive and is implicitly adopted in paper proofs of liveness such as Bravo et al. [2022]. However, continuous objects are difficult to formalize in proof checkers. Therefore, in this work, we replace it with a discrete model that approximates

the continuous behavior. We explain how this model is derived. Without loss of generality, let us assume that the timeout duration δ is $c\Delta$ where c is a positive integer.

Let x be the time duration elapsed since the most recent reset call and $y = c\Delta - x$. Then y is a continuous variable that decreases linearly as time flows. When y reaches 0, the timer sends out a timeout signal. Now instead of focusing on y, we consider its approximate value $t = \lfloor y/\Delta \rfloor$.

We observe that t only changes discretely. If reset is called at timepoint T, then before timepoint $T + \Delta$ we have $t = c - 1$, and it decreases by 1 at timepoints $T + \Delta, T + 2\Delta, \cdots$. The timer sends out its timeout signal at timepoint $T + c\Delta$.

We can picture the discrete changes of t as being triggered by an external *elapse* signal. Formally, we make the timer maintain an internal variable *local_rem_time*. When reset is called, it sets *local_rem_time* $= c - 1$. When it receives an elapse signal, it decreases *local_rem_time* by 1. After *local_rem_time* reaches 0 and the elapse signal is received again, it delivers a timeout. Algorithm 4 shows the formal pseudocode.

Algorithm 4 The Discrete Timer Model

```
1:  Assume timeout duration δ = cΔ.
2:  initialize:
3:      timer_enabled ← true
4:      local_rem_time ← c − 1
5:  upon Reset():
6:      timer_enabled ← true
7:      local_rem_time ← c − 1
8:  upon Elapse():
9:      if timer_enabled then
10:         if local_rem_time > 0 then
11:             local_rem_time ← local_rem_time − 1
12:         else
13:             timer_enabled ← false
14:             Timeout()
```

To use this model in liveness proofs, we also have to specify its live traces. We first formally characterize live traces of a timer using timed traces:

Definition 4.1. A non-Zeno timed trace of a discrete timer is live if:

(1) Before GST, *Reset*() and *Elapse*() can be called arbitrarily;
(2) Within the time interval $[GST, GST + \Delta)$, either *Reset*() or *Elapse*() is called at least once;
(3) After the first *Reset*() or *Elapse*() event after GST, if an *Elapse*() event e exists at timepoint t, then there exists a *Reset*() or *Elapse*() event e' at timepoint $t - \Delta$, and between e, e' there is no other event in the trace;
(4) If a *Reset*() or *Elapse*() event occurs at timepoint $t \geq GST$, and no event occurs within the interval $(t, t + \Delta)$, then there exists a *Reset*() or *Elapse*() event at timepoint $t + \Delta$.

To model the timer in our segmented trace formalism, we notice the following patterns:

Lemma 4.2. *In a live timed trace of a timer, within each interval* $[T, T + \Delta)$ *with* $T \geq GST$*, we have:*

(1) Elapse() is called at most once;
(2) After Reset() is called Elapse() is not called;
(3) Either Elapse() or Reset() is called at least once.

Proof: See Appendix B.

We thus combine Lemma 4.2 with Lemma 2.8 to define the live traces of a network system.

Definition 4.3. A segmented trace $(\tau_0, \tau_1, \cdots)$ of a network system is live if:

(1) τ_0 can be any valid finite trace;
(2) Each segment between τ_i and τ_{i+1} satisfies the patterns in Lemma 4.2 for each individual timer object;
(3) Messages already sent in τ_i are delivered at least once in τ_{i+1}, provided both the sender and the recipient are non-faulty.

$$
\begin{aligned}
Vote \triangleq\ & EMVote(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method) \\
| \ & CVote(\mathbb{N}_{id} * \mathbb{N}_{round}) \\
CacheCert \triangleq\ & EMCert(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method * List(EMVote)) \\
| \ & CCert(\mathbb{N}_{id} * \mathbb{N}_{round} * List(CVote)) \\
Request \triangleq\ & EMReq(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method * TimeoutCert) \\
| \ & EMReq(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method * CCert) \\
| \ & CReq(\mathbb{N}_{id} * \mathbb{N}_{round} * EMCert) \\
Timeout \triangleq\ & Timeout(\mathbb{N}_{id} * \mathbb{N}_{round} * Option(EMCert)) \\
TimeoutCert \triangleq\ & TimeoutCert(\mathbb{N}_{round} * List(Timeout))
\end{aligned}
$$

Fig. 8. Message space of Jolteon.

4.2 Unpipelined Jolteon

In Gelashvili et al. [2022], the Jolteon consensus protocol was described in its pipelined form. Here we consider an unpipelined form. The pipelined form is considered in Section 5 and Appendix D.

Message Space. As shown in Fig. 8, the message space of Jolteon consists of five kinds of messages, which we call $Vote$, $CacheCert$, $Request$, $Timeout$, and $TimeoutCert$. Requests, votes, and cache certificates are subdivided into *EM* type and *C* type. This naming shows the correlation between Jolteon and the ADO model. In Jolteon, *ECache* and *MCache* are created simultaneously in a single phase, and *EMCert* serves as the certificate for these caches; *CCache* is created in a second phase, using *CCert* as its certificate. In the original description [Gelashvili et al. 2022], *CCert* and $TimeoutCert$ are called *QC* and *TC* respectively.

Constraints on Byzantine Processes. In theory, byzantine processes are allowed to send "any message." However, it is common practice to use cryptographic primitives such as digital signatures to constrain their behaviors. We impose two constraints on byzantine processes, called *cryptographic* constraint and *semantic constraint*.

As shown in Fig. 8, each message except $TimeoutCert$ contains a sender ID field (*id*). Also, some messages may embed other messages, like $TimeoutCert$ embedding $Timeout$ messages. Our cryptographic constraint takes a Dolev-Yao-like approach [Dolev and Yao 1983]: byzantine participants may only fill their own IDs in the sender field, and they may only embed already existing messages; however, we give them access to every existing message in the network, regardless of whether they are intended recipients or not.

```
Record node_state := {
    node_local_round : nat;
    node_local_rem_time : nat;
    node_leader_phase : (* Enum type *);
    node_voter_phase : (* Enum type *);
    node_commit_round : nat;
    node_recv_votes : list Vote;
    node_recv_emcache : list EMCert;
    node_recv_timeouts : list TimeoutMsg;
}
```

Fig. 9. State variables of non-faulty process.

In addition to cryptographic validity, we enforce a semantic validity rule. For each kind of message, we define a decidable property on its content that must be satisfied. For example, a cache certificate must embed a quorum of votes supporting that cache. Since these properties are decidable, the non-faulty processes simply call the decision procedure and discard the message if the test fails. This allows us to ignore messages that are not semantically valid, and simplify the proof. See Appendix C for details.

State Space. Fig. 9 shows a simplified view of the internal state of non-faulty processes. Although proposers and voters are logically separate, they are implemented in the same process. The field most relevant to liveness is *local_round*, which dictates which round the process currently participates in. The *commit_round* field records the highest round in which the process has cast a commit vote ($CVote$). It corresponds to the $qc_{high}.r$ variable in the original description. The *local_rem_time* field is the remaining time of the timer object, as discussed previously. The other fields are progress indicators for leaders and voters, and buffers for received messages. Although the buffers are shown as lists here, they are implemented as finite maps from process IDs to messages, and we keep only one message per ID.

Algorithm 5 Unpipelined Jolteon Protocol

Assume $local\_round = r$
▷ Invoke phase
as leader:
    $cert \leftarrow CCert$ or $TimeoutCert$ of round $r - 1$
    if $cert$ is $CCert$ **then**
        $parent\_round \leftarrow r - 1$
    else
        $parent\_round \leftarrow \max_{timeout \in cert} timeout.emcert.round$ (0 if $timeout.emcert = \bot$)
    Choose $method$ from client requests
    Broadcast $EMReq(id, r, parent\_round, method, cert)$
    Collect $EMVote(\_, r, parent\_round, method)$ from a quorum of voters
    $emcert \leftarrow EMCert(id, r, parent\_round, method, votes)$
as voter:
    Wait for $EMReq(\_, r, p, m, cert)$
    Send $EMVote(id, r, p, m)$
▷ Commit phase
as leader:
    Broadcast $CReq(id, r, emcert)$
    Collect $CVote(\_, r)$ from a quorum of voters
    $ccert \leftarrow CCert(id, r, votes)$
    Send $emcert, ccert$ to external client and executors
as voter:
    Wait for $CReq(\_, r, emcert)$
    Store $emcert$, set $commit\_round$ to r
    Send $CVote(id, r)$
▷ Pacemaker
upon timeout:
    $cert \leftarrow EMCert$ of round $commit\_round$, $\bot$ if never sent any $CVote$
    Broadcast $Timeout(local\_round, cert)$
upon receive a quorum of $Timeout$ with $round \geq local\_round$:
    Send $TimeoutCert(local\_round, timeouts)$ to oneself and $leader\_at(local\_round + 1)$

Operation of Jolteon. Algorithm 5 is a summary of our implementation of Jolteon. Each round has two phases, which we call Invoke and Commit. During the Invoke phase, the leader broadcasts a request that contains a $CCert$ or $TimeoutCert$ of the previous round, along with a client method

of its own choice. This corresponds to a simultaneous pull and invoke in ADO. In the second phase the leader rebroadcasts the votes received, and the voters store the method. This corresponds to a push in ADO.

When a process receives a timeout signal from the timer, it broadcasts a *Timeout* message and no longer produces votes for the current round. The *Timeout* message contains the current *local_round*. A quorum of *Timeout*, each of round at least *local_round* (they do not need to be of the same round), is used to build a *TimeoutCert*. Any non-faulty process that receives a *CCert* or *TimeoutCert* should forward it to the next leader. This ensures the leader will also enter the new round within Δ. The pacemaker described in Algorithm 5 corresponds to part (c) of Fig. 1, which is sufficient for our refinement proof. We also implemented a version with pacemaker improved to part (d) of Fig. 1. See Appendix C for details.

The timer is reset when and only when the process increases its *local_round*. The process enters round $r >$ *local_round* in one of the following situations:

(1) A *TimeoutCert* or *CCert* of round $r - 1$ is received;
(2) A valid *Request* of round r is received;
(3) A *Timeout* message that embeds an *EMCert* with $emcert.round = r$ is received.

4.3 Proving Safety and Liveness of Jolteon

We proved both the safety and liveness of Jolteon by constructing a refinement between Jolteon and LiDO. The proof was done in three steps:

(1) We construct a refinement between Jolteon and ADO (Definition 3.2), which derives the safety of Jolteon from the safety of ADO;
(2) For each network state z, we define its abstract pacemaker state, which consists of *round* and *rem_time*, and prove that each network step either does not change these values or changes them in accordance with one action of the abstract pacemaker; this proves that Jolteon refines LiDO;
(3) We prove that all live traces of Jolteon (Definition 4.3) refine live traces of LiDO.

In Appendix C, we present the full details of the proof. Here we present the overall structure and discuss some of its interesting details.

Layered Safety Refinement. The refinement mapping itself is straightforward to define. We map a proposer broadcasting *EMReq* to calling *Pull* at ADO level. Since *ECache* and *MCache* are created in a single phase, we map building *EMCert* to an atomic sequence of creating *ECache*, calling *Invoke*, and creating *MCache*. Broadcasting *CReq* and building *CCert* are mapped to calling *Push* and building *CCache*. If a proposer enters the next round without collecting enough votes for its request, we map it to creating *TCache*.

The hard part is to show that the image of every valid network trace is a valid ADO trace. It is possible to prove this theorem in a single shot. However, the proof would be quite complex and involve dozens of mutually dependent invariants. Instead, we introduced two intermediate layers called **Server** and **Voting** (Fig. 4), which allowed us to reduce proof complexity by proving some invariants on a simpler, more abstract layer. Each lower layer is a transition system that captures more information about the network state but is also more deeply tied to implementation details.

The informal idea of safety proof is as follows. Each *CacheCert* and *TimeoutCert* is backed by a quorum of votes or timeouts. For every pair of a *CCert* of round r and a *TimeoutCert* of round $r' \geq r$, at least one non-faulty voter has voted for both. The *CVote* must have been produced before the *Timeout*. Hence, the highest *EMCert* embedded in the *TimeoutCert* must be of round at least r. Since the *parent_round* of any *EMReq* of round $r' + 1$ must come from either a *CCert* or

a $TimeoutCert$ of round r', we see that the new leader must observe all committed methods. We now decompose the above argument, so that we only deal with one key invariant at a layer.

At **Server** layer, the events we capture are proposers building *Request* and *CacheCert* messages, and the pacemaker building $TimeoutCert$ messages. The proposers, as well as the pacemaker, are modeled as threads running on a shared-memory system. Each thread can observe all existing messages, and may atomically create a single new message. The invariant we enforce at this layer is that for every pair of a *CCert* of round r and a $TimeoutCert$ of round $r' \geq r$, the $TimeoutCert$ must embed an *EMCert* of round at least r.

At **Voting** layer, we additionally capture voters sending $Vote$ and $Timeout$ messages. Again, the voters are threads on a shared-memory system and can observe all existing messages. We enforce that non-faulty voters cannot make conflicting votes. This means they cannot make two different $EMVote$ in a single round, they cannot make $CVote$ of round r after sending $Timeout$ of round $r' \geq r$, etc. It is then easy to prove that the Voting layer refines the Server layer, through the quorum-overlap argument.

The **Network** layer implements the proposer, voter, and timer threads in our message-passing model. The messages must now be explicitly delivered to each process. Each voter maintains its own bookkeeping and decides whether to produce a vote upon receiving a request. To show that the Network layer refines the Voting layer, we prove that whenever a voter decides to produce a vote, the relevant safety invariant is respected.

A Subtle Safety Issue. Although the proof outlined above seems intuitive, there are many subtle details. Here we present one example. Suppose that $leader\_at(r)$ enters round r by receiving a quorum of timeouts of round $r-1$. According to Algorithm 5, it should find the highest *EMCert* embedded within the timeouts. It is possible that an *EMCert* of round $r+1$ has already been created. If so, a byzantine process could include it in a timeout of round r. In this case, when the request succeeds, the leader would have to set $parent\_round = r+1$, which violates ADO safety rules.

The above situation would not actually happen. The reason is that if an *EMCert* of round $r+1$ exists, then a quorum of voters have already entered round $r+1$, and so will not vote on the request of round r. However, this argument is not easy to formalize using invariants. Instead, we adopt a much simpler solution: we make non-faulty processes reject $Timeout$ with $timeout.emcert.r > timeout.r$. This ensures that in every valid $TimeoutCert$ of round r, the highest embedded *EMCert* can be of round at most r, which eliminates the difficult case described above.

Liveness Refinement. Fig. 10 presents an overview of our layered liveness proof. We first proved that the pacemaker mechanism satisfies the protocol-independent assumptions (Definition 3.3). Then we decomposed the time allocated to each round into 3 steps. In the first step, the leader receives a *CCert* or $TimeoutCert$ from the previous round and enters the new round. Then the leader completes the two phases of a round. Each phase is further decomposed into two stages: the voters receive the request, and the leader receives the votes.

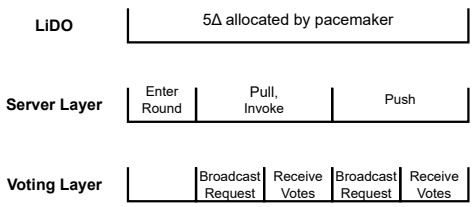

Fig. 10. Liveness of Jolteon

More specifically, we first define for each network state z the corresponding abstract pacemaker states $round(z)$ and $rem\_time(z)$.

Definition 4.4. Let H be the set of non-faulty processes. For each valid network state z, define:

(1) $round(z) = \max_{p \in H} p.local\_round$;

Table 1. Proof effort of unpipelined Jolteon, in lines of Coq.

Part	Lines	Purpose
LiDO Interface	321	Define LiDO object
Safety proof	272	Prove safety of LiDO
Server layer		Add QCs/TCs into view
Spec	185	
Invariants	56	
Refinement	115	
Voting layer		Add votes/timeouts into view
Spec	353	
Invariants	194	
Refinement	275	
Network layer		Model the network system
Spec	939	
Invariants	604	
Refinement	693	
Liveness proof		Prove liveness refinement
Liveness of LiDO	357	
Voting layer to LiDO layer	523	
Network model to Voting layer	953	

(2) $rem\_time(z) = \min_{p \in H, p.local\_round = round(z)} p.local\_rem\_time$.

That is, we take $round(z)$ to be the maximum round ever entered by any non-faulty process. Among the processes currently participating in $round(z)$, we take the minimum value of the timers' remaining time as $rem\_time(z)$. It is easy to see that under this definition, the pacemaker simulates a logical clock. We leave the details in Appendix C.

We then proved that the leader can commit a method within the allocated time. We present one example of proving progress within a single round.

Lemma 4.5. *Let z, z' be the system states at timepoints $T, T + \Delta$. If $round(z) = r$, $leader\_at(r)$ is non-faulty and waits upon $Pull(r)$, $rem\_time(z) \geq 1$, then in state z' either there exists an EMCert of round r, or all non-faulty voters have voted for the same parent_round and method in round r.*

Proof: Since "there exists an *EMCert* of round r in state z'" is a decidable property (by deciding over every existing message in the state), we do a case analysis on it. If such an *EMCert* exists, then we are done. If not, then the leader must still be waiting upon $Pull(r)$. By the GST assumption, the request message must have been delivered to all non-faulty voters.

We then look at the progress indicator of each non-faulty voter. Since $rem\_time(z) \geq 1$, none of the non-faulty voters will timeout within Δ. Hence, when each voter receives the request, it will send a vote, unless it has already received another *EMReq* or *CReq*. The latter case is impossible since the non-faulty leaders do not send conflicting requests.

Proof Effort. Table 1 shows the proof effort for Jolteon. Defining the models and proving the safety property of Jolteon took around 4,000 lines. Proving its liveness took around 1,800 lines.

5 PIPELINED JOLTEON

Pipelining is an optimization technique in consensus protocol design that reduces one phase from each round. It was introduced in HotStuff [Yin et al. 2019] and adopted in several following works, including Jolteon. It is, in fact, how Jolteon was originally presented.

Pipelining works by merging the Commit phase of each round with the Invoke phase of the next round. While pipelining improves latency when there are no byzantine faults [Yin et al. 2019], the fact that committing a method requires the cooperation of two consecutive leaders weakens the liveness guarantee of the protocol. This issue has been studied in Giridharan et al. [2023]. Nevertheless, in the $3f + 1$ rotating-leader setting, one can show there must be at least two consecutive non-faulty leaders by a counting argument. If every non-faulty leader is sandwiched by byzantine leaders then the proportion of byzantine processes must be at least $1/2$ instead of $1/3$.

Verifying pipelined protocols is more challenging than unpipelined protocols. This is because the liveness of pipelining is tied to the round change mechanism. Proving the liveness of each single round is not enough. We also have to analyze the cooperation of consecutive leaders and potential byzantine interference.

We have completed a safety and liveness proof of pipelined Jolteon. This shows our approach can be applied to systems with non-trivial optimizations. The details of our implementation and proof are in Appendix D. Here we present the modifications to the liveness proof.

We observe that the liveness of pipelined Jolteon essentially consists of two parts. First, each non-faulty leader can create an *MCache* on its own. Second, under suitable conditions, two consecutive non-faulty leaders can cooperate to commit the *MCache*.

The "suitable conditions" of the second part are a bit tricky. The first leader initiates pipelining by sending its *EMCert* message to the second leader. On the other hand, the pacemaker may also send a *TimeoutCert* to the second leader. To make pipelining successful, the second leader must receive the *EMCert* before any *TimeoutCert* messages. We implement this by requiring that the first leader must send its *EMCert* soon enough: when it sends out *EMCert* we must have $rem\_time \geq 1$. This implies that no *TimeoutCert* will be created within Δ, so the next leader must build its own request using the received *EMCert*.

Our liveness theorem is as follows:

Theorem 5.1. *In every infinite segmented trace of pipelined Jolteon, let $r = round(\tau_0)$, then for every $r' > r$ such that both $leader\_at(r'), leader\_at(r' + 1)$ are non-faulty, eventually an MCache and a CCache of r' are created.*

Proof Effort. The safety proof effort remains almost the same. The liveness proof grew slightly more complex and required around 2,000 lines.

6 EXPERIMENTAL EVALUATION

To show that our Coq specification is realistic and faithful to runnable code, we extracted the network layer specification of unpipelined Jolteon into executable OCaml code. The code specifies messages to be exchanged among different nodes and abstractions for sending and receiving messages but lacks implementations of the network primitives and the timer. We manually glued the network abstraction to OCaml's libraries that realize TCP/UDP-based communications through a shim layer and added a timer that triggers timeout events when the round does not advance within a threshold. Still, the main logic comes from the unmodified extracted code.

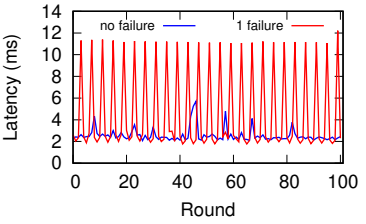

Fig. 11. Latency measurements.

We evaluate the code on a research cloud with a four-node setting. Fig. 11 shows a series of latency measurements to increment the round either by committing a method or by timing out. Without any failed or Byzantine nodes, the system exhibits an average latency of 2.56 ms to commit a request. With a single failed or Byzantine node that hinders making progress, it takes an average of

4.45 ms to advance the round (timeout is set to 10 ms). The system is not optimized for performance and does not include pipelining, but the experiment shows that our specification is realistic, the code maintains liveness under failure, and the execution exhibits comparable performance (i.e., 1 ms overhead under steady state) to other verified PBFT Rahli et al. [2018] and non-verified BFT-SmaRt implementations [Bessani et al. 2014].

7 RELATED WORK

Theoretical Solutions to Byzantine Consensus Liveness. Maintaining the liveness of byzantine consensus protocols has been traditionally considered a difficult problem. The original PBFT thesis [Castro 2001] did not give a formal liveness proof, although they had a semi-formal safety proof. The problem is that byzantine participants may attempt to force an early round-change, and honest participants need to correctly deal with the messages they send.

HotStuff [Yin et al. 2019] first proposed to use an independent component called pacemaker to control round-change so that each round gets allocated sufficient time to commit methods. However, the pacemaker of HotStuff is relatively unusual. The participants may enter new rounds without observing QC or TC of previous rounds. Therefore, HotStuff had to use exponential backup to ensure that, eventually, the participants would enter the same round. Its liveness dynamics are difficult to analyze. Jolteon [Gelashvili et al. 2022] was then proposed as a variant of HotStuff that reverts to a pacemaker with the all-to-all broadcast of timeout messages. The Cogsworth pacemaker [Naor et al. 2021] was proposed as a way to avoid all-to-all broadcast needed in Jolteon. It has been incorporated into a new version of HotStuff [Malkhi and Nayak 2023]. While our work has only inspected the pacemaker of Jolteon, we expect that most of the pacemaker designs in the literature can be captured and analyzed by our approach.

Bravo et al. [2022] proposed a theory of *synchronizers*, which are objects that control the round-change of each process but are completely independent of other parts of the protocol. They showed that it can applied to a number of different protocol designs. However, synchronizers are not band-aids that magically repair broken protocols. To apply the synchronizer to a protocol requires changes to the protocol itself, and indeed a large part of Bravo et al. [2022] is showing that the modified protocol still satisfies safety and liveness. This shows that synchronizers do not replace the need for a formal framework for safety and liveness proofs. We also observe that it is unclear how to apply synchronizers to pipelined protocols, as pipelining relies on a fast path for round change, which synchronizers currently do not provide.

Verifying Safety and Liveness of Consensus Protocols. A large number of frameworks for verifying the correctness of consensus protocols have been proposed in the literature. Figure 2 gives a comparison between our work and existing approaches. The figure shows a clear pattern: verifying safety is relatively easy, but verifying liveness is a lot harder. Especially for byzantine consensus protocols, all existing liveness results only work for fully asynchronous or synchronous protocols.

A number of projects have aimed at verifying the safety properties of byzantine consensus protocols similar to HotStuff [Yin et al. 2019]. These include Velisarios [Rahli et al. 2018], Carr et al. [2022], and QTree [Cirisci et al. 2023]. In particular, the Velisarios proof uses a logic-of-events approach, which constructs a causal ordering of events and proves safety by induction on this ordering, with which our safety refinement proof bears similarity. However, recording only causal ordering does not provide enough information to establish liveness. For partially synchronous protocols, one also needs temporal ordering, which our segmented-trace formalism addresses.

Carr et al. [2022] suggests that one proves a weak version of liveness called *plausible liveness*, which essentially means that one can always extend any valid execution to commit some data. This notion is inadequate in an adversarial environment: the byzantine adversary may actively

Table 2. Comparison between consensus verification projects.
*: The liveness proof does not cover partially-synchronous protocols.

	Byzantine	Safety	Liveness	Executable	Refinement
IronFleet [Hawblitzel et al. 2015]	×	✓	✓	✓	✓
Verdi [Wilcox et al. 2015]	×	✓	×	✓	✓
PSync [Drăgoi et al. 2016]	×	✓	✓	✓	✓
Taube et al. [2018]	×	✓	×	✓	×
Velisarios [Rahli et al. 2018]	✓	✓	×	✓	✓
Adore [Honoré et al. 2022]	×	✓	×	✓	✓
Carr et al. [2022]	✓	✓	×	×	×
QTree [Cirisci et al. 2023]	✓	✓	×	×	✓
Padon et al. [2017]	×	✓	✓*	×	×
Berkovits et al. [2019]	✓	✓	✓*	×	×
Losa and Dodds [2020]	✓	✓	✓*	×	×
Bertrand et al. [2022]	✓	✓	✓*	×	×
LiDO (this work)	✓	✓	✓	✓	✓

delay successful commit. Another issue is the protocol may selectively ignore certain requests. Our notion of liveness guarantees every proposer may always commit some method of its own choice.

IronFleet [Hawblitzel et al. 2015] and PSync [Drăgoi et al. 2016] are the only results we are aware of that cover liveness and can be connected to executable code. Both works only cover benign consensus. PSync uses the Heard-Of model, and the verified code is coupled to a pacemaker. The pacemaker component is not mechanically verified. IronFleet explicitly models timers, heartbeats, and other factors. The model is very comprehensive, but the accompanying proofs are equally verbose. Our methodology results in proofs with a more transparent structure and better reusability.

Padon et al. [2017] proposed a liveness-to-safety reduction approach that allows verifying the liveness of protocols in first-order logic. It has been extended to byzantine protocols in Berkovits et al. [2019]; Losa and Dodds [2020], but has so far not been applied to partially synchronous protocols. Our work has shown that it is easy to both capture network dynamics using safety properties, and decompose SMR liveness into safety requirements on the network. However, automating our proofs in model checkers is future work.

AdoB [Honoré et al. 2024] is a recent variant of ADO that supports reasoning about benign and byzantine faults in a unified way. The main difference between AdoB and LiDO is that AdoB is an atomic model, whereas LiDO defines a concurrent but linearizable object. This has significant implications for liveness reasoning. Refinement proofs for AdoB linearize each valid network trace into a valid atomic trace of AdoB. In doing so, it reorders network events and eliminates important temporal information. For example, even if the trace τ_1 is a prefix of τ_2, there is no general relation between their linearized traces τ_1' and τ_2'. Consequently, although AdoB claims to have a liveness proof, it does not support liveness refinement like our LiDO model does: live traces of the network model cannot be directly correlated to live traces of AdoB.

Consensus Beyond Partial Synchrony. In this work, we have only considered mechanizing liveness proof of partially synchronous protocols with a fixed set of participants. In practice, public blockchains often demand byzantine consensus algorithms supporting dynamic or open membership. There are a number of works proposing protocol designs that work under this new setting [Buterin et al. 2020; D'Amato et al. 2023]. Also, Thomsen and Spitters [2021] have mechanized a liveness proof for Nakamoto-style Proof-of-Stake (PoS) consensus under a synchronous setting. In the future, we plan to extend our theory to cover open-membership protocols.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 2019285, 1763399, 2313433, and 2118851, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

ARTIFACT-AVAILABILITY STATEMENT

The artifact accompanying this paper is available on Zenodo [Qiu et al. 2024].

REFERENCES

Mark Abspoel, Thomas Attema, and Matthieu Rambaud. 2021. Brief Announcement: Malicious Security Comes for Free in Consensus with Leaders. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 195–198. https://doi.org/10.1145/3465084.3467953

Idan Berkovits, Marijana Lazić, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 245–266. https://doi.org/10.1007/978-3-030-25543-5_15

Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:24. https://doi.org/10.4230/LIPIcs.DISC.2022.10

Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 355–362. https://doi.org/10.1109/DSN.2014.43

Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. 2022. Liveness and Latency of Byzantine State-Machine Replication. In *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA (LIPIcs, Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:19. https://doi.org/10.4230/LIPIcs.DISC.2022.12

Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and Casper. arXiv:2003.03052 [cs.CR]

Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. 2022. Towards Formal Verification of HotStuff-Based Byzantine Fault Tolerant Consensus in Agda. In *NASA Formal Methods*, Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez (Eds.). Springer International Publishing, Cham, 616–635. https://doi.org/10.1007/978-3-031-06773-0_33

Miguel Castro. 2001. *Practical Byzantine Fault Tolerance*. Ph. D. Dissertation. Massachusetts Institute of Technology. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf

Berk Cirisci, Constantin Enea, and Suha Orhun Mutluergil. 2023. Quorum Tree Abstractions of Consensus Protocols. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 337–362. https://doi.org/10.1007/978-3-031-30044-8_13

Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. 2022. Byzantine Consensus Is $\Theta(n^2)$: The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony!. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:21. https://doi.org/10.4230/LIPIcs.DISC.2022.14

Francesco D'Amato, Joachim Neu, Ertem Nusret Tas, and David Tse. 2023. Goldfish: No More Attacks on Ethereum?! arXiv:2209.03255 [cs.CR]

D. Dolev and A. Yao. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory* 29, 2 (1983), 198–208. https://doi.org/10.1109/TIT.1983.1056650

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *Journal of the ACM* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security*, Ittay Eyal and Juan Garay (Eds.). Springer International Publishing, Cham, 296–315. https://doi.org/10.1007/978-3-031-18283-9_14

Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. 2023. BeeGees: Stayin' Alive in Chained BFT. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing* (Orlando, FL, USA) *(PODC '23)*. Association for Computing Machinery, New York, NY, USA, 233–243. https://doi.org/10.1145/3583668.3594572

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. 2021. Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 97 (oct 2021), 31 pages. https://doi.org/10.1145/3485474

Wolf Honoré, Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2024. AdoB: Bridging Benign and Byzantine Consensus with Atomic Distributed Objects. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 109 (apr 2024), 30 pages. https://doi.org/10.1145/3649826

Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. Adore: Atomic Distributed Objects with Certified Reconfiguration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 379–394. https://doi.org/10.1145/3519939.3523444

Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-Fairness for Byzantine Consensus. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 451–480. https://doi.org/10.1007/978-3-030-56877-1_16

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Klaus Kursawe. 2020. Wendy, the Good Little Fairness Widget: Achieving Order Fairness for Blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (New York, NY, USA) *(AFT '20)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3419614.3423263

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

Leslie Lamport. 2005. *Real Time is Really Simple*. Technical Report MSR-TR-2005-30. 72 pages. https://www.microsoft.com/en-us/research/publication/real-time-is-really-simple/

Andrew Lewis-Pye. 2022. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. *CoRR* abs/2201.01107 (2022). arXiv:2201.01107

Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020) (OpenAccess Series in Informatics (OASIcs), Vol. 84)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:9. https://doi.org/10.4230/OASIcs.FMBC.2020.9

Dahlia Malkhi and Kartik Nayak. 2023. Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT. Cryptology ePrint Archive, Paper 2023/397. https://eprint.iacr.org/2023/397

Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2021. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems* 1, 2 (oct 22 2021). https://doi.org/10.21428/58320208.08912a03

Oded Naor and Idit Keidar. 2020. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. In *34th International Symposium on Distributed Computing (DISC 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:17. https://doi.org/10.4230/LIPIcs.DISC.2020.26

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035 Festschrift for John C. Reynolds's 70th birthday.

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2017. Reducing Liveness to Safety in First-Order Logic. *Proc. ACM Program. Lang.* 2, POPL, Article 26 (dec 2017), 33 pages. https://doi.org/10.1145/3158114

Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. 2024. *Artifact for PLDI 2024 paper # 290: LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs.* Yale University, New Haven, USA. https://doi.org/10.5281/zenodo.10909272

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 619–650. https://doi.org/10.1007/978-3-319-89884-1_22

Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (dec 1990), 299–319. https://doi.org/10.1145/98163.98167

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (dec 2017), 30 pages. https://doi.org/10.1145/3158116

Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 113–129. https://doi.org/10.1145/3600006.3613172

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. *SIGPLAN Not.* 53, 4 (jun 2018), 662–677. https://doi.org/10.1145/3296979.3192414

Søren Eller Thomsen and Bas Spitters. 2021. Formalizing Nakamoto-Style Proof of Stake. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–15. https://doi.org/10.1109/CSF51468.2021.00042

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. *SIGPLAN Not.* 50, 6 (jun 2015), 357–368. https://doi.org/10.1145/2813885.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

A SAFETY THEORY OF ADO

Here we present the safety properties of the ADO model. The ADO model has been defined in Section 3.

A.1 Explanation of the Cache Creation Preconditions

We first explain the cache creation preconditions shown in Fig. 6.

ECache Creation. An *ECache* of round r represents the consensus log branch that the leader of round r receives when it enters round r. The $parent\_round$ variable represents the round number of the latest entry of that branch. The conditions are:

- $p < r$: The log branch must come from a previous round. It cannot come from a later round, otherwise there would be cycles in the cache tree.
- $p = 0 \vee \Sigma[p].mcache \neq \bot$: Either the leader receives an empty log (represented by $p = 0$), or the latest entry is a valid *MCache*.
- $\Sigma[r].ecache = \bot$: There can be only one *ECache* per round.
- $\forall r', r' < r \Rightarrow (p \geq r' \vee \Sigma[r'].ccache = \bot)$: For every previous round r', either no *CCache* was created in round r', or the latest entry of the consensus log branch is from at least round r'. This ensures the branch will always contain all committed methods in previous rounds.

MCache Creation. An *MCache* of round r represents the method proposed by the leader of round r in that round. The conditions are:

- $\Sigma[r].ecache \neq \bot$: The leader must have already received a consensus log branch.
- $\Sigma[r].mcache = \bot$: There can be only one *MCache* per round.

CCache Creation. A *CCache* of round r represents that the method proposed in round r has been successfully committed, along with all previous methods in that branch of consensus log. The conditions are:

- $\Sigma[r].mcache \neq \bot$: The leader must have already proposed a method in round r.
- $\forall r', r' > r \Rightarrow (\Sigma[r'].ecache = \bot \vee \Sigma[r'].ecache.parent\_round \geq r)$: For every future round r', either the leader of round r' has not yet entered round r' and received a consensus log branch, or the latest entry of the received branch is from at least round r. This ensures that we do not retroactively commit methods that are not seen by future leaders, thus creating a fork situation.

A.2 Some Simple Lemmas of ADO

We state a few simple invariants about the cache tree Σ. They can be easily proved by induction on the cache creation trace.

Lemma A.1. *Once a cache node c is added to Σ, it is never removed.*

Lemma A.2. *If a cache node c is in Σ, then $parent(c) \in \Sigma$.*

Lemma A.3. *Every ECache in Σ satisfies $ecache.round > ecache.parent\_round$.*

Lemma A.4. *Every cache node c in Σ satisfies $c.round \geq parent(c).round$, taking $Root.round = 0$.*

Lemma A.5. *There is at most one ECache, one MCache, and one CCache in each round.*

Lemma A.6. *If $\Sigma[r].ccache \neq \bot$, then for every $r' > r$, if $\Sigma[r'].ecache \neq \bot$, then $\Sigma[r'].ecache.parent\_round \geq r$.*

Lemma A.7. *The cache tree Σ is always well-defined.*

A.3 Lemmas on the Cache Tree

For any two cache nodes $c_1, c_2 \in \Sigma$, we say c_2 is a descendant of c_1, if there exists a path from c_1 to c_2 in the cache tree, including the case $c_1 = c_2$.

We now make some simple observations:

Lemma A.8. *If c_2 is a descendant of c_1, then $c_1.round \leq c_2.round$.*

Lemma A.9. *If c_3 is a descendant of both c_1, c_2, then either c_2 is a descendant of c_1, or c_1 is a descendant of c_2.*

A.4 The ADO Safety Invariant

The key invariant maintained by the ADO object is the following:

Lemma A.10. *If $\Sigma[r].ccache \neq \bot$, then every cache node c with $c.round > r$ is a descendant of $\Sigma[r].mcache$.*

Proof: By induction on $c.round$. Since the parent of an *MCache* is the *ECache* of the same round, and the parent of a *CCache* is the *MCache* of the same round, we only need to care about the *ECache* of each round.

In the base case $c.round = r + 1$, then we must have $r \leq c.parent\_round < r + 1$, which means $c.parent\_round = r$ and $parent(c) = \Sigma[r].mcache$.

In the inductive case, we still have $r \leq c.parent\_round < c.round$. Let $c' = \Sigma[c.parent\_round].mcache$, then c is a descendant of c', and by inductive hypothesis c' is a descendant of $\Sigma[r].mcache$, so c is a descendant of $\Sigma[r].mcache$.

We are now ready to prove the safety theorem of ADO:

Lemma A.11. *In the cache tree there exists a path starting from Root and contains all committed MCache.*

Proof: Let $c_1, c_2, \cdots$ be the list of all committed *MCache* in Σ, ordered by round number. Since the cache tree is well-formed, there exists a path from *Root* to c_1. Then we only have to show that there always exists a path from c_i to c_{i+1}.

Since both c_i, c_{i+1} are committed, there exists two *CCache* nodes d_i, d_{i+1}, such that d_i is a descendant of c_i, and d_{i+1} is a descendant of c_{i+1}.

If $d_i.round = d_{i+1}.round$ then d_i is a descendant of both c_i, c_{i+1}. Since $c_i.round < c_{i+1}.round$, we infer that c_{i+1} is a descendant of c_i.

Otherwise, we have either $d_i.round < d_{i+1}.round$, or $d_i.round > d_{i+1}.round$. If $d_i.round < d_{i+1}.round$, then d_{i+1} is a descendant of $\Sigma[d_i.round].mcache$, which in turn is a descendant of c_i. But d_{i+1} is also a descendant of c_{i+1}. Hence c_{i+1} is a descendant of c_i. The case where $d_i.round < d_{i+1}.round$ is completely analogous.

By induction, we can always construct a path that passes through all committed *MCache*.

B LIVENESS REQUIREMENTS OF THE TIMER OBJECT

In Section 4.1, we introduced the timer object (Algorithm 4). We now prove Lemma 4.2, which is key to defining the live traces of timers under segmented traces.

For reader's convenience, we copy over the relevant definition.

Definition B.1. A live timed trace of a discrete timer is a valid timed trace that satisfies the following conditions:

(1) Before GST, $Reset()$ and $Elapse()$ can be called arbitrarily;
(2) Within the time interval $[GST, GST + \Delta)$, either $Reset()$ or $Elapse()$ is called at least once;

(3) After the first $Reset()$ or $Elapse()$ event after GST, if an $Elapse()$ event e exists at timepoint t, then there exists a $Reset()$ or $Elapse()$ event e' at timepoint $t - \Delta$, and between e, e' there is no other event in the trace;
(4) If a $Reset()$ or $Elapse()$ event occurs at timepoint $t \geq GST$, and no event occurs within the interval $(t, t + \Delta)$, then there exists a $Reset()$ or $Elapse()$ event at timepoint $t + \Delta$.

Lemma B.2. *In a live timed trace of a timer, within each interval* $[T, T + \Delta)$ *with* $T \geq GST$ *we have:*
(1) $Elapse()$ *is called at most once;*
(2) After $Reset()$ *is called* $Elapse()$ *is not called;*
(3) Either $Elapse()$ *or* $Reset()$ *is called at least once.*

Proof: If a $Reset()$ or $Elapse()$ event occurs at timepoint x, then the next $Elapse()$ event would not occur until timepoint $x + \Delta$. This proves the first two patterns.

To prove the third pattern, first consider the special case $T = GST + k\Delta$ where k is integer. Definition B.1 guarantees that at least one event occurs within the interval $[GST, GST + \Delta)$, which proves the base case $k = 0$.

To prove the inductive case, assume that at least one event occurs within the interval $[GST + k\Delta, GST + (k + 1)\Delta)$. Let e be the last event in this interval and let t be the timepoint of e. Then no event occurs in the interval $(t, GST + (k + 1)\Delta)$. If at least one event occurs in the interval $[GST + (k + 1)\Delta, t + \Delta)$ then we are done. Otherwise, Definition B.1 guarantees that at least one event occurs at timepoint $t + \Delta$, which finishes the inductive case.

Finally, we consider the general case $T = GST + k\Delta + x$ with $0 < x < \Delta$. If the interval $[GST + k\Delta + x, GST + (k+1)\Delta)$ contains at least one event, then we are done. Otherwise the interval $[GST + k\Delta, GST + k\Delta + x)$ contains at least one event. Let e be the last event in this interval and let t be the timepoint of e. Then there is no event in the interval $(t, GST + (k + 1)\Delta)$. Therefore, if no event occurs in the interval $[GST + (k + 1)\Delta, t + \Delta)$, then at least one event occurs at timepoint $t + \Delta$. This finishes the analysis.

C DETAILS OF UNPIPELINED JOLTEON

C.1 Semantic Validity of Jolteon Messages

Here we explain the semantic validity condition we impose on each kind of message.

Vote. All vote messages must have $round \geq 1$. This is because rounds are numbered from 1. $EMVote$ additionally requires $parent\_round \leq round$.

CacheCert. The requirements are:

- $round \geq 1$: Round number must be valid.
- $id = leader\_at(round)$: Only the leader of round r can create cache certificates of round r.
- The embedded $Vote$ must have matching type and data. This means $EMCert$ can only embed $EMVote$ with same $round$, $parent\_round$, and $Method$. $CCert$ can only embed $CVote$ with same $round$.
- The embedded $Vote$ must come from a quorum of voters.

Request. The requirements are:

- $round \geq 1$: Round number must be valid.
- $id = leader\_at(round)$: Only the leader of round r can create requests of round r.
- If $round = r$, $EMReq$ must embed either a $TimeoutCert$ or a $CCert$ of round $r - 1$. If $CCert$ is embedded, $parent\_round = r - 1$. If $TimeoutCert$ is embedded, $parent\_round$ is equal to the round number of the highest embedded $EMCert$ in the $TimeoutCert$. If no $EMCert$ is embedded, take $parent\_round = 0$.

- If $round = r$, $CReq$ must embed an $EMCert$ of round r.

Timeout. The requirements are:

- $round \geq 1$: Round number must be valid.
- $round \geq emcert.round$: If a non-faulty process receives an $EMCert$ of round r, and $local\_round < r$, then it should enter round r immediately. Hence when a process times out and broadcasts a timeout message, it should not embed an $EMCert$ of round $r' > r$.

TimeoutCert. The requirements are:

- $round \geq 1$: Round number must be valid.
- If $round = r$, then every embedded $Timeout$ message satisfies $round \geq r$.
- If $round = r$, then every embedded $EMCert$ (indirectly through $Timeout$ messages) satisfies $emcert.round \leq r$. If a non-faulty process receives a $Timeout$ that embeds an $EMCert$ of round r, and $local\_round < r$, then it should enter round r immediately. Hence when the process builds a $TimeoutCert$ of round r, it should not embed an $EMCert$ of round $r' > r$. This ensures that when $leader\_at(r+1)$ receives the $TimeoutCert$, it would not see an entry that "comes from the future."
- The embedded $Timeout$ messages come from a quorum of voters.

C.2 Proof of Liveness Lemmas

Lemma C.1. *For a non-faulty process p, if $p.local\_rem\_time = k$, then it will not send any $Timeout$ message within $k\Delta$.*

Proof: A non-faulty process sends $Timeout$ only when local timer times out, but local timer elapse occurs at most once in each period of Δ, and each event decreases $p.local\_rem\_time$ by 1, so it will not timeout within $k\Delta$.

Lemma C.2. *If $rem\_time(z) = k$ then no $TimeoutCert$ of $round(z)$ will be built within $k\Delta$.*

Proof: A $TimeoutCert$ embeds a quorum of $Timeout$ messages, with at least $f+1$ of them coming from non-faulty processes. However, $rem\_time(z) = k$ implies none of them will send $Timeout$ message within $k\Delta$.

Lemma C.3. *If any non-faulty process p enters round $r > 1$, then either $leader\_at(r)$ has already entered round r, or p has received a $CCert$ or $TimeoutCert$ of round $r-1$.*

Proof: By case analysis on the different ways a process may enter round r.

(1) A $TimeoutCert$ or $CCert$ of round $r-1$ is received.
This is exactly the second condition.
(2) A valid *Request* of round r is received.
A non-faulty leader builds a request of round r only after entering round r, so the first condition is satisfied.
(3) A $Timeout$ message that embeds an $EMCert$ with $emcert.round = r$ is received.
A non-faulty leader builds an $EMCert$ of round r only after entering round r, so the first condition is satisfied.

Lemma C.4. *If any non-faulty process p enters round $r > 1$, then some non-faulty process has previously entered round $r-1$. By induction, every round $r' < r$ has been entered by some non-faulty process.*

Proof: By Lemma C.3, the first process to enter round r must have received a $TimeoutCert$ or a $CCert$ of round r. A $CCert$ embeds a quorum of $CVote$, but a non-faulty process produces a $CVote$ of round r only after entering round r.

Similarly, a $TimeoutCert$ embeds a quorum of $Timeout$ messages of round $r' \geq r$. Hence some non-faulty process has entered some round $r' \geq r$. By induction hypothesis, we see some non-faulty process must have previously entered round r.

Lemma C.5. *The live traces of unpipelined Jolteon satisfy Definition 3.3 and Definition 3.5.*

Proof: We start from the easy pieces:

(1) Between τ_i and τ_{i+1}, $Elapse()$ is called at most once
This follows from each network timer object elapses at most once between τ_i and τ_{i+1}.

(2) If $rem\_time(\tau_i) > 0$, between τ_i and τ_{i+1} if $Elapse()$ is not called then $round$ is increased at least once.
If $round$ did not increase, this means no non-faulty process entered a round higher than $round(\tau_i)$. Then the timer of every process that is currently in $round(\tau_i)$ elapses once, so $rem\_time(z)$ decreases by 1.

(3) If $round(\tau_i) = r$, both $leader\_at(r), leader\_at(r+1)$ are non-faulty, $leader\_at(r)$ has called $StartNext(r)$, then $round(\tau_{i+1}) \geq r + 1$.
A non-faulty leader calling $StartNext(r)$ corresponds to sending a $CCert$ to the next leader. Since it will be delivered within Δ, and the new leader will enter round $r + 1$ upon receiving it, we have $round(\tau_{i+1}) \geq r + 1$.

(4) A non-faulty leader calls $StartNext(r)$ only after a $CCache$ in round r is created.
A non-faulty leader calls $StartNext(r)$ by sending a $CCert$ message of round r, which corresponds to a $CCache$ of round r.

(5) If $round(\tau_i) = r$, $leader\_at(r)$ is non-faulty, but no $ECache$ of round r has been created, then either $leader\_at(r)$ is currently waiting upon $Pull(r)$, or it will call $Pull(r)$ before the end of τ_{i+1}.
If $ECache$ has not been created, this means $leader\_at(r)$ either has not yet entered round r, or it is still collecting $EMVote$. In the former case, Lemma C.3 implies it will enter within Δ. In the latter case, it has already called $Pull(r)$ and is waiting for response.

(6) If $round(\tau_i) = r$, $leader\_at(r)$ is non-faulty, $rem\_time \geq 2$, and the leader is waiting upon a call $Pull(r), Invoke(r, m), Push(r)$, then that call will succeed before the end of τ_{i+2}.
Making each call corresponds to broadcasting the corresponding request message, and collect a quorum of votes. Within Δ the request will reach every non-faulty voter, and within another Δ the votes will be received by the leader. The condition $rem\_time \geq 2$ guarantees that no non-faulty process will send $Timeout$ messages within 2Δ, by Lemma C.2.

(7) When a non-faulty leader receives $Success$ for $Pull(r)$, it immediately chooses m and calls $Invoke(r, m)$; similarly it immediately calls $Push(r)$ after $Invoke(r, m)$ succeeds.
Sending out $CReq$ is part of the message handler for $EMVote$. In our model it is performed atomically with receiving the last vote message of a quorum set. Therefore, the proposer immediately calls $Push(r)$ after $Invoke(r, m)$ succeeds.

The most complex part is showing that when $rem\_time(z)$ reaches 0, there is a constant C such that $round(z)$ increases within $C\Delta$. We give a proof with large C that is easy to formalize. Proofs with smaller C can be constructed with more detailed analysis, or by implementing a more complex pacemaker, but this proof suffices for our purpose.

Assume that the leader schedule is fair, meaning each proposer becomes the leader once every $3f + 1$ rounds. This is typical in rotating-leader protocols. Suppose that z, z' are the network states at the beginning and end of a period of Δ. Let $r = round(z)$. Since $round$ increases in unit steps, for every round $r' < r$ there must exist a previous state z'' with $round(z'') = r'$. When $round$ reaches r', the leader of r' must enter round r' within Δ. Hence we can prove that in state z', every non-faulty process should be in some round $r' > round(z) - (3f + 1)$.

By Theorem 3.6, we set the timeout duration of each timer to 8Δ. Then we observe:

Lemma C.6. *Within every period of 9Δ, at least one non-faulty process will enter a higher round.*

Proof: Let z be the network state at the beginning of the period, and let x be the minimum round r any non-faulty process is still participating in state z. If no non-faulty process enters a higher round within 8Δ, then every non-faulty process will broadcast a $Timeout$ message of round $r' \geq x$, and within Δ everyone will receive a quorum of $Timeout$ messages. Hence every process still in round x will enter round $x+1$.

Since there are $2f+1$ non-faulty processes, by pigeonhole principle, if we wait for a period of $9(3f+1)(2f+1)\Delta$, at least one non-faulty process must have entered some round $r \geq round(z)+1$. This proves non-faulty processes must keep entering new rounds.

C.3 Jolteon with Improved Pacemaker

The pacemaker of Algorithm 5 (Line 27–31) corresponds to part (c) of Fig. 1. We have also implemented a version of unpipelined Jolteon that improves the pacemaker to part (d) of Fig. 1. The improved pacemaker is shown as follows.

Algorithm 6 Improved Pacemaker for Jolteon Implementation

```
1: upon timeout:
2:     cert ← EMCert of round commit_round, ⊥ if never sent any CVote
3:     Broadcast Timeout(local_round, cert)
4: upon receive a quorum of Timeout with round ≥ local_round:
5:     Send TimeoutCert(local_round, timeouts) to oneself and leader_at(local_round + 1)
6: upon receive f + 1 Timeout with round > local_round:
7:     Enter round local_round + 1
```

The intuition is that, if there are $f+1$ $Timeout$ messages with $round > local\_round$, then at least one of them comes from a non-faulty process. That process must have already entered some round $r' > local\_round$.

The safety proof remains almost the same. However, with this change we can derive a better latency bound. If one voter enters round r by receiving a $TimeoutCert$, then among the embedded $Timeout$ messages, at least $f+1$ come from non-faulty processes. These will be received by every voter within Δ. Hence all voters will enter round $r-1$ within Δ. On the other hand if it enters round r by receiving a $CCert$, then at least $f+1$ voters have already entered round $r-1$. This shows that the round number difference between the voters cannot be too large.

D DETAILS OF PIPELINED JOLTEON

D.1 Changes to the Jolteon Protocol

The system model is the same as described in Section 4.1. The message space is reduced and shown in Fig. 12. The internal state of non-faulty processes is the same as Fig. 9. The pipelined Jolteon protocol is described in Algorithm 7. There is now only one phase per round. The changes from Algorithm 5 are shown in blue. The conditions for a non-faulty process to enter round r are:

(1) A $TimeoutCert$ or $EMCert$ of round $r-1$ is received;
(2) A valid $Request$ of round r is received;
(3) A $Timeout$ message that embeds an $EMCert$ with $emcert.round = r$ is received.

$$
\begin{aligned}
Vote &\triangleq EMVote(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method) \\
CacheCert &\triangleq EMCert(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method * List(EMVote)) \\
Request &\triangleq EMReq(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method * TimeoutCert) \\
&\mid EMReq(\mathbb{N}_{id} * \mathbb{N}_{round} * \mathbb{N}_{parent\_round} * Method * EMCert) \\
Timeout &\triangleq Timeout(\mathbb{N}_{id} * \mathbb{N}_{round} * Option(EMCert)) \\
TimeoutCert &\triangleq TimeoutCert(\mathbb{N}_{round} * List(Timeout))
\end{aligned}
$$

Fig. 12. Message space of pipelined Jolteon.

Algorithm 7 Pipelined Jolteon Protocol

Assume $local\_round = r$
▷ Invoke phase
as leader:
  $cert \leftarrow$ *EMCert* or $TimeoutCert$ of round $r - 1$
  if $cert$ is $EMCert$ **then**
    $parent\_round \leftarrow r - 1$
  else
    $parent\_round \leftarrow \max_{timeout \in cert} timeout.emcert.round$ (0 if $timeout.emcert = \bot$)
  Choose $method$ from client requests
  Broadcast $EMReq(id, r, parent\_round, method, cert)$
  Collect $EMVote(\_, r, parent\_round, method)$ from a quorum of voters
  $emcert \leftarrow EMCert(id, r, parent\_round, method, votes)$
  Send $emcert$ to $leader\_at(r + 1)$; also send to $leader\_at(r - 1)$ if $parent\_round = r - 1$
as voter:
  Wait for $EMReq(id, r, p, m, cert)$
  if $p = r - 1$ **then**
    Extract *EMCert* of round $r - 1$ from $cert$
    Store *EMCert*, set $commit\_round$ to $r - 1$
  Send $EMVote(id, r, p, m)$
▷ Pacemaker
upon timeout:
  $cert \leftarrow EMCert$ of round $commit\_round$, $\bot$ if never sent any $EMVote$ with $p = r - 1$
  Broadcast $Timeout(id, local\_round, cert)$
upon receive a quorum of $Timeout$ with $round \geq local\_round$:
  Send $TimeoutCert(local\_round, timeouts)$ to oneself and $leader\_at(local\_round + 1)$

D.2 Changes to the Refinement Proof

Safety Refinement. We still map broadcasting *EMReq* to calling *Pull*, and creating *EMCert* to creating *ECache* and *MCache*. In pipelined Jolteon the leader does not perform the commit phase by itself. It delegates the task to the next leader and simply waits for the result. Therefore, we map the network event of the current leader receiving *EMCert* from the next leader to the ADO event of calling *Push* and create *CCache*.

The overall proof architecture is the same as that for unpipelined Jolteon. However, there is now no explicit *CVote*. Instead, an *EMVote* with $parent\_round = round - 1$ serves as a commit vote for $parent\_round$. Therefore, we prove instead the following invariants:

Lemma D.1. *If a non-faulty process produces both an EMVote of round r with $parent\_round = r-1$, and a Timeout of round $r' \geq r$, then the Timeout embeds an EMCert of round $r'' \geq r-1$.*

Lemma D.2. *If an EMCert of round r is created with $parent\_round = r-1$, then every TimeoutCert of round $r' \geq r$ embeds an EMCert of round $r'' \geq r-1$.*

From these lemmas we can prove that creating *CCache* in round $r-1$ after creating *ECache* in round r with $parent\_round = r-1$ is always safe.

Liveness Refinement. Since pipelined protocols provide weaker liveness guarantees than unpipelined ones, we have to tune the ADO-specific liveness requirements in the model.

Definition D.3. For pipelined protocols we impose the following liveness requirements:

(1) A non-faulty leader calls $StartNext(r)$ only after a *MCache* in round r is created.
(2) If $round(\tau_i) = r$, $leader\_at(r)$ is non-faulty, but no *ECache* of round r has been created, then either $leader\_at(r)$ is currently waiting upon $Pull(r)$, or it will call $Pull(r)$ between τ_i and τ_{i+1};
(3) When $leader\_at(r)$ receives *Success* for $Pull(r)$, it immediately chooses some m and calls $Invoke(r, m)$;
(4) If $round(\tau_i) = r, rem\_time(\tau_i) \geq 2$, and $leader\_at(r)$ is waiting upon a *Pull* or *Invoke* call of round r, it will succeed before the end of τ_{i+2};
(5) If $round(\tau_i) = r$, an *MCache* of round r has been created, both $leader\_at(r), leader\_at(r+1)$ are non-faulty, then $round(\tau_{i+1}) \geq r+1$.
(6) If $round(\tau_i) < r$ or $round(\tau_i) = r \wedge rem\_time(\tau_i) \geq 2$, but $round(\tau_{i+1}) \geq r+1$, and $leader\_at(r+1)$ is non-faulty, then an *ECache* of round $r+1$ with $parent\_round = r$ is created before the end of τ_{i+3}.
(7) If at the end of τ_i, an *ECache* of round r with $parent\_round = r-1$ exists, and both $leader\_at(r), leader\_at(r-1)$ are non-faulty, then a *CCache* of round $r-1$ is created before the end of τ_{i+1}.

Like the requirements for unpipelined protocols, these rules provide a breakdown of how a method is committed in a pipelined protocol. The first four rules guarantee that each non-faulty leader can individually create an *MCache*. The last three rules show how two non-faulty leaders cooperate to commit an *MCache*.

The proofs for the first three rules are exactly the same as in the unpipelined case. Here we focus on proofs for the pipelined case:

(1) If $round(\tau_i) = r$, an *MCache* of round r has been created, both $leader\_at(r), leader\_at(r+1)$ are non-faulty, then $round(\tau_{i+1}) \geq r+1$.
Creating *MCache* corresponds to creating *EMCert* at network level. Then $leader\_at(r)$ immediately sends it to the next leader, and it will be delivered within Δ. Hence $round(\tau_{i+1}) \geq r+1$.
(2) If $round(\tau_i) < r$ or $round(\tau_i) = r \wedge rem\_time(\tau_i) \geq 2$, but $round(\tau_{i+1}) \geq r+1$, and $leader\_at(r+1)$ is non-faulty, then an *ECache* of round $r+1$ with $parent\_round = r$ is created before the end of τ_{i+4}.
If $round(\tau_i) < r$ or $rem\_time(\tau_i) \geq 2$ then no *TimeoutCert* of round r would be created within 2Δ. On the other hand if $round(\tau_{i+1}) \geq r+1$ then some non-faulty process must have entered round $r+1$. That process must have received an *EMCert* of round r. Since it would forward the *EMCert* to $leader\_at(r+1)$, the new leader must receive it before the end of τ_{i+2}. It could not receive any *TimeoutCert* before the end of τ_{i+2}. Therefore, it will embed the *EMCert* into its *EMReq*, and the *ECache* it subsequently creates must have $parent\_round = r$.

(3) If at the end of τ_i, an *ECache* of round r with $parent\_round = r - 1$ exists, and both $leader\_at(r), leader\_at(r-1)$ are non-faulty, then a *CCache* of round $r - 1$ is created before the end of τ_{i+1}.
After $leader\_at(r)$ creates the *EMCert*, it sends the certificate to $leader\_at(r-1)$, which is delivered within Δ. The previous leader uses the certificate to confirm that its proposal has been committed, creating the *CCache*.

We can now prove that:

Theorem D.4. *In every infinite segmented trace of pipelined Jolteon, let $r = round(\tau_0)$, then for every $r' > r$ such that both $leader\_at(r), leader\_at(r+1)$ are non-faulty, eventually a method is committed in round r.*

First we can show that every non-faulty proposer can independently build *EMCert*. The proof is almost the same as Theorem 3.6. By choosing a suitable timeout duration, we can further show that by the time the proposer builds *EMCert*, the abstract pacemaker must have $rem\_time \geq 2$. Now the proposer sends the *EMCert* to the next leader. By assumption it will rebroadcast that *EMCert* and thus commit the method.

Received 2023-11-16; accepted 2024-03-31