Anoma Research Topics | TECHNICAL REPORT

Heterogeneous Paxos 2.0: the Specs

Aleksandr Karbyshev [a] **and Isaac Sheff** [a]

[a]Heliax AG

* *E-Mail: karbyshev@mailbox.org, isaac@heliax.dev*

Abstract

We present *Heterogeneous Paxos 2.0* (HP2.0), an improved version of Heterogeneous Paxos consensus algorithm (HP). In a nutshell, HP2.0 simplifies the algorithm logic, reduces bandwidth usage, and enables a more efficient implementation.
HP2.0 is compatible with the requirements of HP and satisfies the same correctness properties. A formal specification of HP2.0 in TLA$^+$ is available as a separate software artefact, with a formal proof of the key safety property of *Agreement* in TLAPS.
This report provides an accessible account of HP2.0, of the design space in which it exists, and of the design choices that led us to our current system.

Keywords: Consensus ; Distributed Algorithm ; Heterogeneous Paxos ;

(Received: June 17, 2024; Version: December 4, 2024)

Contents

1. Introduction

We present *Heterogeneous Paxos 2.0* (HP2.0): an improved, simpler, and more efficient version of Heterogeneous Paxos consensus protocol (HP) [SWvRM20]. The assumptions and guarantees of HP2.0 are similar to those of HP, but we simplify the algorithm logic and reduce communication complexity, thus improving efficiency.

A formal specification of HP2.0 in TLA$^+$ [Lam02] is available in the Typhon repository [Ano]. The TLA$^+$ spec has a formally verified proof of Agreement, a key safety property [Ano]. Here, rather than focusing on proofs, we attempt to explain the protocol in more reader-friendly terms.

This report is intended as the definitive "source of truth" for the definition of HP2.0. However, as we continue to integrate and prove more optimizations and simplifications, HP2.0 remains a work in progress.

1.1. Original Heterogeneous Paxos

This report assumes readers are familiar with the original HP technical report [SWvRM20], and is intended to explain the improvements we have made for HP2.0.

Original HP generalizes the Paxos consensus protocol [Lam98] for a setting with arbitrary quorums of acceptors, tolerating mixed crash and byzantine failures, with each learner tolerating *different* failure scenarios. The motivation, roles for participants, and trust model of HP2.0 are identical to those in HP [SWvRM20, §1–4, 8].

1.2. Why Heterogeneous Paxos 2.0?

HP2.0 improves upon HP by being both simpler and more efficient. The simplifications not only make the protocol easier to formally specify in TLA$^+$, but also facilitate proving things about it and streamline its implementation. The differences between HP and HP2.0 are outlined in Section 2.

HP2.0 admits a substantially more efficient implementation than HP, with no loss in power. For example, calculating which byzantine acceptors are *caught* in a message could naively require exponential time for HP. The Byzantine behaviour detection modification of HP2.0 (Section 2.4) enables an implementation that can perform this check in linear time.

2. Differences From Original Heterogeneous Paxos

The HP2.0 consensus algorithm results from five substantial improvements to HP, which we describe in the remainder of this section.

2.1. Broadcast Primitive

HP2.0 assumes a *broadcast* primitive, which is responsible for sending messages to each acceptor and learner. For the liveness property to hold, *broadcast* must guarantee that if a message is received by one honest acceptor (i.e., an acceptor that is both safe and live), then that message is eventually received by all acceptors. It does not require ordering to be preserved.

HP2.0 assumes a partially synchronous network [DLS88], where after some unknown global stabilization time (GST), all messages arrive within some (unknown but fixed) latency bound. After GST, the broadcast guarantee must deliver messages to all honest recipients within an (unknown) finite bound dependent on network latency. We do not require any particular message ordering guarantees.

In the HP algorithm, *broadcast* was implemented by having each acceptor echo well-formed messages to all other acceptors. In contrast, HP2.0 leaves the exact implementation of *broadcast* flexible, allowing for multiple possible approaches. One such implementation, as used in HP, involves all acceptors echoing all received messages to all other acceptors and learners. A more bandwidth-efficient implementation, however, would require acceptors to explicitly request any missing messages when a received message references one they do not recognize.

2.2. 2a Messages

Instead of sending a *2a*-message for each learner as in HP [SWvRM20], we send a single *2a* (non-proposal) message associated with a set of learners. Conceptually, this is similar to sending a set of *2a*-messages, but in practice, it is more efficient, both to send and to track with *refs* (Section 3.3).

2.3. One Message In, at Most One Message Out

With the broadcast primitive and the *2a*-messages, we can substantially simplify our protocol: In fact, we can remove all recursion and broadcast at most one message for each message received. This entails another minor change:

instead of each actor receiving its own message in the same atomic action that it sends it (messages are broadcast, so actors receive their own messages), they receive them in some future action, just like any other message. This change, in turn, may increase implementation efficiency by breaking up atomic actions into smaller schedulable pieces.

2.4. Byzantine Behaviour Detection

Each message specifies the previous message from the same sender (Section 3.3). This makes it much easier to detect certain kinds of Byzantine behaviour: two messages referencing the same parent form a proof of misbehaviour. In contrast, the original protocol called for comparing transitive history sets [SWvRM20]. This change makes detecting Byzantine behaviour much easier to implement without sacrificing any guarantees.

2.5. Logic changes

In some cases, we were able to simplify Heterogeneous Paxos by removing unnecessary constraints and checks. Our formal proof shows that the streamlined HP2.0 protocol retains the same safety conditions (Agreement and Validity) as the original. In each case, simplification can only improve liveness: all acceptor actions allowed without the simplification remain allowed with the simplification. Termination guarantees therefore remain unchanged.

We make three such simplifications: in the definition of Buried, the definition of Well-Formed, and the acceptor algorithm.

2.5.1. Buried

In HP2.0, to *bury* a *2a*-message x, the acceptor must have seen a different *2a*-message with a higher ballot and a different value. In contrast, HP required that a quorum of acceptors observe such a message [SWvRM20, Definition 28]. Upon analysing the formal safety proof for HP, we found that the quorum was only used to establish the existence of a single higher ballot *2a*-message with a different value. We have formally proven that the quorum condition is unnecessary for burying an older *2a*-message.

By eliminating this unnecessary constraint, HP2.0 consensus algorithm can achieve faster progress than HP in certain scenarios, as the acceptors no longer need to wait for redundant *2a*-messages to propagate.

2.5.2. Well-Formed

In HP2.0, we introduce a generalized definition of the well-formedness condition for *2a*-messages (see [SWvRM20, Assumption 29] for the original well-formedness assumption). Specifically, we have abstracted all restrictions into a single *WellFormed2a* predicate, with the only remaining concrete constraint being that the set of references is not empty (Definition 20).

We demonstrate that the algorithm remains safe regardless of the concrete instantiation of *WellFormed2a*, even when it is set to a constant True predicate. However, we advise a more restrictive predicate instantiation (as discussed in Section 3.7), which rules out certain redundant *2a*-messages, including those which HP would otherwise permit. In particular, when an acceptor sends two *2a*-messages for the same ballot in a row, the later one must feature more learners than the first. If this condition is not met, the second message does not enable any new actions. All recipients would have to process the first message anyway (due to causal order of message receipt), and it enables the same actions.

2.5.3. Acceptor Algorithm

With our modifications to Well-Formed and streamlined message structure, the space of allowed messages is actually very narrow: there isn't much a byzantine acceptor can do, besides forget history. As a result, our acceptor algorithm becomes very straightforward: receive messages in causal order, and at each message receipt, send a well-formed message if possible (messages must reference the previous message sent, and all messages received since then). The one exception to this is that $1a$-messages for old ballots are completely ignored: there is no need to process old proposals.

This acceptor algorithm is very generic: it moves almost all of the consensus protocol logic into the computation of *well-formedness*. Indeed, almost all protocols with all-to-all communication could be described this way (although it wouldn't necessarily be efficient).

3. Specification

3.1. Network Model

We assume the *closed-world* distributed system consisting of a fixed finite set of *acceptors* $\mathbb{A}$, a fixed finite set of proposers $\mathbb{P}$ and a fixed finite set of *learners* $\mathbb{L}$. We denote by $\mathcal{S} \subseteq \mathbb{A}$ a set of *safe*, non-Byzantine, acceptors that follow the protocol.

We assume that message broadcast is *reliable*, i.e., every message sent or received by a correct (i.e., safe and live) acceptor or proposer is eventually received by all live acceptors and learners[1]. The delivery order of sent messages does not have to be preserved.

3.2. Learner Graph

We use the notion of *learner graph* introduced in [SWvRM20]. We recap its formal definition below.

Let $\mathbb{L}$ be a fixed finite set of learners, and $\mathbb{A}$ a fixed finite set of acceptors.

[1] As learners are never required to send messages, all learners are trivially live.

- Nodes of learner graph are elements $\alpha \in \mathbb{L}$.
- Each learner α is labelled with a set $Q_\alpha \subseteq 2^{\mathbb{A}}$. The elements of Q_α are quorums $q \subseteq \mathbb{A}$. We assume that each Q_α is closed upwards, i.e., for any $q' \supseteq q \in Q_\alpha$, we have $q' \in Q_\alpha$.
- For any pair $\alpha, \beta \in \mathbb{L}$, there is a graph edge labelled with a set of quorums, α–$\beta \subseteq 2^{\mathbb{A}}$, called a *safe set* of α and β. We assume that any safe set α–β is closed upwards, i.e., for any $q' \supseteq q \in \alpha$–$\beta$, we have $q' \in \alpha$–β.
- The graph is undirected, i.e., α–$\beta = \beta$–α.

Definition 1 (Condensation)**.** *We say that the learner graph is* ***condensed*** *if for all* $\alpha, \beta, \gamma \in \mathbb{L}$

$$\alpha\text{–}\beta \cap \beta\text{–}\gamma \subseteq \alpha\text{–}\gamma$$

Definition 2 (Validity)**.** *We say that the learner graph is* ***valid*** *if for any* $s \in \alpha$–β*,* $q \in Q_\alpha$*, and* $r \in Q_\beta$ *we have* $s \cap q \cap r \neq \emptyset$*.*

Definition 3 (Entanglement)**.** *We say that learners* α *and* β *are* ***entangled*** *if the set of safe acceptors* $\mathcal{S}$ *belongs to the learners' safe set.*

$$\mathit{Entangled}(\alpha, \beta) \stackrel{\text{def}}{=} \mathcal{S} \in \alpha\text{–}\beta$$

3.3. Protocol Message Structure

As far as message encoding "on the wire" is concerned, there are two types of messages: **proposer messages**, also called 1a-messages, and **acceptor messages**. The encoding of these must be distinct, allowing recipients to identify the type of the received message.

3.3.1. Properties of All Protocol Messages

1. Each protocol message x carries a cryptographic signature that uniquely identifies the message signer, denoted by $\mathit{Sig}(x)$, with $\mathit{Sig}(x) \in \mathbb{A} \cup \mathbb{P}$.
2. Each protocol message has a unique hash, which other messages can use to reference it.

3.3.2. Properties of Proposer Messages

1. For each proposer message x, $\mathit{Sig}(x)$ must be a proposer: a participant authorized to generate new proposals.
2. Each proposer message x has a field $x.\mathit{val}$ containing a proposed value.

3. Each proposer message x has a field $x.bal$: a natural number specific to this proposal—its *ballot* number. We assume that ballots are value-specific: $x.bal = y.bal$ implies $x.val = y.val$. One way to implement this is to include the hash of the value in the (least significant bits of the) ballot.

3.3.3. Properties of Acceptor Messages

1. Each acceptor message x carries a finite list of references to some other messages, $x.refs$. In practice, each reference is represented by a hash of the referenced message. Acceptors remember previously received messages (*known_messages*), allowing them to resolve references.

2. Each acceptor message x carries a distinguished reference, $x.prev$, to a previous message signed by the same signer, identified by $Sig(x)$. If x is the first message sent by the sender, $x.prev$ contains a special non-reference value $\perp$.

3.4. Definitions

Definition 4 (1b)**.** *Acceptor message x is called a 1b-*message*, denoted as $x : 1b$, if its references contain a proposer message. Formally:*

$$x : 1b \stackrel{\text{def}}{=} \exists y \in x.refs.\, y : 1a$$

Definition 5 (2a)**.** *Acceptor message x is called a 2a-*message*, denoted as $x : 2a$, if its references do* not *contain a proposer message. Formally:*

$$x : 2a \stackrel{\text{def}}{=} \forall y \in x.refs.\, \neg y : 1a$$

We extend *Sig* over sets of messages to mean the set of signers of those messages:

Definition 6 (Message set signers)**.** *For any set of messages M:*

$$Sig(M) \stackrel{\text{def}}{=} \{Sig(m) \mid m \in M\}$$

Definition 7 (Transitive references)**.**

$$Tran(x) \stackrel{\text{def}}{=} \{x\} \cup \bigcup_{m \in x.refs} Tran(m)$$

Definition 8 (Message 1a)**.**

$$Get1a(x) \stackrel{\text{def}}{=} \operatorname*{argmax}_{m : 1a \in Tran(x)} m.bal$$

Definition 9 (Ballot).

$$B(x) \stackrel{\text{def}}{=} Get1a(x).bal$$

Definition 10 (Value).

$$V(x) \stackrel{\text{def}}{=} Get1a(x).val$$

Definition 11 (Caught acceptors).

$$Caught(x) \stackrel{\text{def}}{=} Sig\left(\left\{ m \in Tran(x) \;\middle|\; \begin{array}{l} \exists m' \in Tran(x). \\ \quad Sig(m) = Sig(m') \\ \wedge\, m \neq m' \\ \wedge\, m.prev = m'.prev \end{array} \right\}\right)$$

Definition 12 (Connected learners). *For any learner α and message x:*

$$Con_\alpha(x) \stackrel{\text{def}}{=} \{\beta \in \mathbb{L} \mid \exists s \in \alpha\text{–}\beta.\, s \cap Caught(x) = \emptyset\}$$

Definitions 13 to 17 are mutually recursive. They are sound because the recursive call is done on descendants of argument messages relative to the message reference relation.

Definition 13 (Buried 2a-messages). *For any learner α, 2a-message x and message y:*

$$\begin{array}{l} Buried_\alpha(x, y) \stackrel{\text{def}}{=} \\ \quad \exists z \in Tran(y).\, z\!:\!2a \wedge \alpha \in lrns(z) \wedge B(z) > B(x) \wedge V(z) \neq V(x) \end{array}$$

Definition 14 (Connected 2a-messages). *For any learner α and message x:*

$$Con2as_\alpha(x) \stackrel{\text{def}}{=} \left\{ m \in Tran(x) \;\middle|\; \begin{array}{l} m\!:\!2a \\ \wedge\, Sig(m) = Sig(x) \\ \wedge\, \exists \beta \in lrns(x). \\ \quad \beta \in Con_\alpha(x) \wedge \neg Buried_\beta(m, x) \end{array} \right\}$$

Definition 15 (Freshness). *For any learner α and 1b-message x:*

$$fresh_\alpha(x) \stackrel{\text{def}}{=} \forall m \in Con2as_\alpha(x).\, V(m) = V(x)$$

Definition 16 (Quorum of message). *For any learner α, 2a-message x:*

$$q_\alpha(x) \stackrel{\text{def}}{=} Sig\big(\{m \in Tran(x) \mid m\!:\!1b \wedge fresh_\alpha(m) \wedge B(m) = B(x)\}\big)$$

Definition 17 (2a-message learners). *For any 2a-message x:*

$$lrns(x) \stackrel{\text{def}}{=} \{\alpha \in \mathbb{L} \mid q_\alpha(x) \in Q_\alpha\}$$

The following *chain* property does not have a direct analogue in HP. It requires that the parent of a message be from the same sender, and included in the messages *refs* field.

Definition 18 (Chain property)**.** *For any message* x*:*

$$ChainRef(x) \stackrel{\text{def}}{=} x.prev \neq \bot \longrightarrow x.prev \in x.refs \land Sig(x.prev) = Sig(x)$$

Definition 19 (Decision)**.** *For any learner* α *and set of messages* S*:*

$$Decision_\alpha(S) \stackrel{\text{def}}{=} Sig(S) \in Q_\alpha \land \forall x, y \in S.\, x{:}2a \land \alpha \in lrns(x) \land B(x) = B(y)$$

Any set of messages S *such that* $Sig(S) \in Q_\alpha$ *is called a* message quorum*. If* $Decision_\alpha(S)$ *holds, we shall write* $V(S)$ *to denote* $V(x)$ *for some message* $x \in S$*.*

Definition 20 (Well-formedness)**.** *For any message* x*:*

$$\begin{aligned}
&WellFormed1b(x) \stackrel{\text{def}}{=} \forall y \in Tran(x).\, x \neq y \land y \neq Get1a(x) \longrightarrow B(y) \neq B(x)\\
&WellFormed(x) \stackrel{\text{def}}{=}\\
&\qquad ChainRef(x)\\
&\quad \land (x{:}1b \longrightarrow WellFormed1b(x))\\
&\quad \land (x{:}2a \longrightarrow x.refs \neq \emptyset \land WellFormed2a(x))
\end{aligned}$$

where WellFormed2a is an arbitrary predicate on messages.[2]

3.5. Protocol

The formal specification of the protocol is formulated in TLA$^+$ [Lam02] in the Typhon repository [Ano]. The acceptor and learner algorithms are formulated in PlusCal language [Lam09], which gets translated to TLA$^+$ code.

For better readability, we present the pseudocode of learner and acceptor algorithms in Figures 1 and 2, respectively. Like TLA$^+$, we specify exactly which actions are *safe* for each actor. This does not rule out unnecessary or repetitive actions. Formally, each action is a *predicate* over state changes: the action is only deemed safe if every part of the predicate holds true. Actions can include other actions.

To ensure liveness, we require *weak fairness* [Lam02]: in any execution trace, if an action is safe for an infinite sequence, it eventually occurs.

Next, we describe the semantics of some particular instructions.

[2] We show in Theorem 22 that the algorithm's *safety* does not depend on the choice of *WellFormed2a* predicate. In particular, it can be chosen to be the constant True predicate. See Section 3.7 for a discussion on the choice of the predicate.

```
Acceptor::init():
  known_messages = {}
  recent_messages = {}
  prev_message = ⊥

Acceptor::receive(m):
  assume m ∉ known_messages
  assume ∀r ∈ m.refs. r ∈ known_messages
  known_messages ∪= {m}

Acceptor::process(m):
  assume WellFormed(m)
  receive(m)
  with z = msg(prev = prev_message, refs = recent_messages ∪ {m}):
    if WellFormed(z):
      recent_messages = {z}
      prev_message = z
      broadcast(z)
    else if ¬(m : 1a):
      recent_messages ∪= {m}
```

Figure 1. Heterogeneous Paxos 2.0 acceptor specification. For all incoming messages, the acceptor calls `Acceptor :: process` message handler. The function `msg` constructs a new *acceptor* message.

```
Learner::init():
  known_messages = {}
  decision = ⊥

Learner::receive(m):
  assume m ∉ known_messages
  assume ∀r ∈ m.refs. r ∈ known_messages
  known_messages ∪= {m}

Learner::process(m):
  assume WellFormed(m)
  receive(m)

Learner::decide():
  with s ⊆ known_messages:
    assume Decision_self(s)
    decision = V(s)
```

Figure 2. Heterogeneous Paxos 2.0 learner specification. For all incoming messages, the learner first calls `Learner :: process` message handler, and then checks whether a decision can be formed based on the set of known messages.

Instruction Semantics.

`broadcast(z)` When called, it sends message z to every learner and acceptor, including the caller. Formally, this would mean that the defined state change is only allowed if the new state includes z in the network.

`assume P` is a synonym of PlusCal's "`when`" instruction [Lam24]. The instruction, defined for Boolean state predicate P, restricts possible executions and should be considered as a guard à la Dijkstra [Dij75]: execution of the function containing "`assume P`" is only possible if the predicate evaluates to true during such a putative execution.

`with x = v: C` is adopted from the PlusCal counterpart [Lam24]: C is executed with x being locally defined as the result of computation of the expression v.

3.6. Protocol Properties

Theorem 21 (Validity)**.** *For any learner α, and set of messages s:*

$$Decision_\alpha(s) \implies \exists x.\, x : 1a \land V(s) = V(x)$$

Proof. Directly follows from the definitions of B, V and *Decision*. □

We formally prove the following key safety property.

Theorem 22 (Agreement)**.** *Let the learner graph of the network be valid and condensed. Let $\alpha, \beta \in \mathbb{L}$ be learners such that Entangled(α, β). For any protocol execution and reachable network state, if $Decision_\alpha(s_\alpha)$ and $Decision_\beta(s_\beta)$ hold in that state, for some message quorums s_α and s_β, then $V(s_\alpha) = V(s_\beta)$.*

Proof. See TLA$^+$ protocol formalization [Ano]. □

3.7. Choice of WellFormed2a Predicate

Although any predicate *WellFormed2a* does not affect protocol safety (Agreement, see Theorem 22), it can affect liveness. If *WellFormed2a* is a constant False predicate, no learner will ever decide. Conversely, if *WellFormed2a* is always True, then acceptors will send a *2a*-message each time they receive one (including their own). In this case, the protocol would technically be live, but acceptors would end up sending an infinite number of unnecessary *2a*-messages, even after all learners decide. To avoid these issues while preserving liveness, we propose the following instantiation of the predicate:

Definition 23.

$$WellFormed2a(x) \stackrel{\text{def}}{=} lrns(x) \neq \emptyset \land (x.prev : 2a \rightarrow lrns(x.prev) \neq lrns(x))$$

The first conjunct of the definition prevents unnecessary *2a*-messages that carry no learner values, as such messages do not contribute to establishing a decision (see Definition 19). The soundness of the second conjunct is based on the following observation: when a correct acceptor sends two *2a*-messages consecutively without sending a *1b*-message in between, both *2a*-messages must have the same ballot number. Therefore, if the newer of the two *2a*-messages contains the same set of learner values as the older one, it does not enable any new actions. Thus, the message is redundant and can be omitted without breaking liveness.

We argue that the proposed definition of *WellFormed2a* preserves liveness—any learner with a safe and live quorum will eventually decide—while ensuring that the number of *2a*-messages a correct acceptor sends in any ballot is bounded by the number of learners.[3]

3.8. Mailbox Layer

The predicates `Acceptor::receive()` and `Learner::receive()` (Figures 1 and 2) define *causal message delivery*, i.e., the message can be received only once and only if all its direct references have already been received. In practice, this logic can be separated out into a special *mailbox* component. This component can be used by both local acceptor and learner, avoiding code duplication and allowing for more efficient message processing.

A mailbox implementation could be integrated into the broadcast implementation (Section 2.1): one mailbox might request missing messages from another when it receives a message m, but has not yet received m's causal dependencies.

4. Future Work

Due to the recursive structure of protocol messages, a naïve implementation of the presented algorithm—utilizing caching of learner values for *2a*-messages—would result in message processing complexity that is polynomial in l and n, where l represents the number of learners in the network and n is the total number of messages received so far.

In ongoing work, we are developing an efficient implementation of HP2.0 using additional data structures achieving a linear time complexity of message processing, $\mathcal{O}(l \cdot n)$. We also plan to formally prove that this implementation correctly realizes the proposed algorithm.

5. Acknowledgements

We would like to thank Murdoch James Gabbay, Christopher Goes, and Jonathan Prieto-Cubides for reviewing this report and providing helpful

[3] A Byzantine acceptor could send a larger, but still finite, number of messages in a ballot.

feedback and suggestions for improvement.

References

Ano. Anoma. Typhon project github repository. https://github.com/anoma/typhon/releases/tag/hpaxos-2.0-art-v4. (cit. on pp. 2, 9, and 11.)

Dij75. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. (cit. on p. 11.)

DLS88. Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. (cit. on p. 3.)

Lam98. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. (cit. on p. 2.)

Lam02. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. (cit. on pp. 2 and 9.)

Lam09. Leslie Lamport. The pluscal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009. (cit. on p. 9.)

Lam24. Leslie Lamport. *A PlusCal User's Manual. C-Syntax Version 1.8*, 2024. (cit. on p. 11.)

SWvRM20. Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers. Heterogeneous paxos: Technical report, 2020. (cit. on pp. 2, 3, 4, and 5.)