Functional Reasoning for Distributed Systems with Failures

HAOBIN NI, Cornell University, USA
ROBBERT VAN RENESSE, Cornell University, USA
GREG MORRISETT, Cornell University, USA

Distributed system theory literature often argues for correctness using an informal, Hoare-like style of reasoning. While these arguments are intuitive, they have not all been foolproof, and whether they directly correspond to formal proofs is in question.

We formally ground this kind of reasoning and connect it to standard formal approaches through language design and meta-analysis, which leads to a functional style of compositional formal reasoning for a class of distributed systems, including cases involving Byzantine faults.

The core of our approach is twin languages: Sync and Async, which formalize the insight from distributed system theory that an asynchronous system can be reduced to a synchronous system for more straightforward reasoning under certain conditions. Sync describes a distributed system as a single, synchronous, data-parallel program. It restricts programs syntactically and has a functional denotational semantics suitable for Hoare-style formal reasoning. Async models a distributed system as a collection of interacting monadic programs, one for each non-faulty node in the system. It has a standard trace-based operational semantics, modeling asynchrony with interleaving. Sync compiles to Async and can then be extracted to yield executable code. We prove that any safety property proven for a Sync program in its denotational semantics is preserved in the operational semantics of its compiled Async programs. We implement the twin languages in Rocq and verify the safety properties of two fault-tolerant consensus protocols: BOSCO and SeqPaxos.

ACM Reference Format:
Haobin Ni, Robbert van Renesse, and Greg Morrisett. 2025. Functional Reasoning for Distributed Systems with Failures. 1, 1 (October 2025), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Distributed system theory often argues for correctness using an informal, Hoare-like style of reasoning [Abraham et al. 2019; Buterin et al. 2020; Kotla et al. 2010; Lamport 1998; Ongaro and Ousterhout 2014; Song and van Renesse 2008]. These arguments are intuitive but are sometimes unsound [Abraham et al. 2017; Michael et al. 2017; Momose and Cruz 2019; Neu et al. 2021]. Their formal implications are also unclear, as the primary approaches to formally reasoning about distributed systems have been trace-based operational semantics and global invariants to handle asynchrony and faults [Alpern and Schneider 1987; Chandy and Lamport 1985; Hawblitzel et al. 2015; Lamport 2002; Ma et al. 2019; Qiu et al. 2025; Wilcox et al. 2015].

In this paper, we formally ground this informal style of reasoning and connect it to the standard approach through language design and meta-analysis. In doing so, we establish a functional denotational semantics for distributed systems, enabling a class of asynchronous systems, potentially with Byzantine faults, to be reasoned compositionally following the syntax of their programs, as functions with non-deterministic outputs.

Authors' Contact Information: Haobin Ni, Cornell University, Ithaca, NY, USA; Robbert van Renesse, Cornell University, Ithaca, NY, USA; Greg Morrisett, Cornell University, Ithaca, NY, USA.

ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
SimpleVote (p : 𝔹@L, x : 𝔹@R)
    : (option 𝔹)@L :=
  let cnt := comm c x 0 (fcnteq p) in
  ret (calc_dec cnt p)
```

(a) SimpleVote in simplified Sync syntax

```
fcnteq (p : 𝔹)(cnt : ℕ)(v : 𝔹) : ℕ :=
  if (p == v) then (cnt + 1) else cnt
calc_dec (cnt : ℕ)(p : 𝔹) : option 𝔹 :=
  if (cnt ≥ n − 2 ∗ f) then (Some p) else None
```

(b) Auxiliary pure function definitions

We propose twin languages, Sync and Async, inspired by distributed system theory [Charron-Bost and Schiper 2009] that an asynchronous system can sometimes be reduced to a synchronous system for reasoning. Sync enables the aforementioned style of reasoning for a class of distributed systems, as if they were synchronous and data-parallel. Async and the relationship between the twin languages ensure that any safety property proven with Sync's functional semantics remains true when the system executes asynchronously as described by Async's standard trace semantics.

We illustrate the problem and our contribution by defining and reasoning about an example program. Figure 1a shows the `SimpleVote` program in simplified Sync syntax. The Sync syntax is *choreographic*, describing the whole system with a single program. Two auxiliary functions are defined separately as pure functions in Figure 1b.

`SimpleVote` aims to describe a distributed system comprising a single leader node and multiple replica nodes that collectively decide on a boolean value based on their local inputs. The program has three steps: first, all replica nodes send their local inputs to the leader; second, the leader node waits until it receives the enough messages from replicas and counts how many of the received values are equal to its local input with `fcnteq`; finally, the leader node computes its local output by checking if the number of equal values reaches a threshold with `calc_dec`.

More concretely, `SimpleVote` assumes two *roles*, i.e., classes of nodes that execute the same code locally: leader and replica, denoted as L and R in the syntax. It has two implicit parameters, n and f, assuming there is a single non-faulty leader node and n replica nodes, with at most f replica nodes being Byzantine and behaving arbitrarily. The first line declares that the program takes in two *distributed* values p and x. p is a single Boolean input to the leader, and x contains a Boolean input for each of the replica nodes. The second line specifies that the return value is an optional Boolean value at the leader node. The following line is a let binding that binds the result of a *communication* action to variable `cnt`. The communication action specifies replica nodes should send their input value x to the leader node, who would wait for at least $n - f$ messages and count the number of messages containing a value equal to p by folding the function (`fcnteq` p) over the list of received messages with the default value 0. The name c in the line uniquely identifies this particular round of communication. The final line specifies that the leader should compute `calc_dec cnt` p to produce an output. In particular, if there are at least $n - 2f$ votes containing p, the leader should return Some p and None otherwise.

A safety property of this program is that when $n > 3f$ and the leader and all correct replicas have the same Boolean value b as their input, the output of the leader must be Some b, no matter the actions of the faulty nodes.

We argue informally in three steps following the syntax of `SimpleVote`. First, because the leader waits for at least $n - f$ messages, among which at most f messages come from faulty nodes, there are at least $n - 2f$ messages received from the correct replica nodes, and all of those contain the value b. Second, by the definition `fcnteq`, this implies `cnt` is at least $n - 2f$. Last, by the definition of `calc_dec`, we derive that the leader's output is Some b.

As we can see, this style of reasoning is more straightforward than devising global invariants that would imply the safety property to be proven, which must cover all possibilities of asynchrony. However, could this be too good to be true? Two questions left to answer are:

(1) How can we make these arguments formally and precisely?

(2) Do we get the same guarantees as if we had done reasoning using invariants on traces?

Our first contribution is to formally ground this style of high-level reasoning. We address (1) by designing Sync and its functional denotational semantics, allowing arguments in the above style to be stated precisely as formal proofs.

Our second contribution is to formally connect to established verification paradigms. We address (2) by proving the adequacy of Sync's semantics for reasoning in regard to a canonical asynchronous model of distributed systems, Async. Async is a low-level language with a standard trace-based operational semantics that models asynchrony as interleaving, which can be reasoned with standard formal verification techniques. More specifically, we prove that for any program p in Sync, any output produced by the traces derived from compiling p to Async, is included in the functional semantics of Sync. This guarantees that any safety properties proven in the Sync semantics also hold for the compilation result in the Async semantics.

Our third contribution is to present a novel framework of formal functional reasoning for the implementation of a class of distributed systems with failures. By combining Sync and Async, one can reason about a distributed system as a composition of non-deterministic functions following its program syntax, while providing strong guarantees about the behavior of the extracted executable code. To showcase this, we have implemented the twin languages in the Rocq proof assistant and verified two consensus protocols as case studies: Bosco, a one-step Byzantine-fault tolerant consensus protocol; and Sequential Paxos, a crash-fault tolerant consensus protocol that is a variant of Paxos [Lamport 1998]. Our more precise reasoning uncovered optimizations that were missed in the original protocols.

For the rest of the paper, Section 2 presents the syntax and semantics of Sync, and Section 3 applies it to verify the two consensus protocols. Section 4 clarifies our assumptions about the network behavior and adversarial node and presents the syntax and semantics of Async, and Section 5 proves the relationship between the twin languages.

2 Syntax and High-Level Semantics for Sync

This section introduces Sync, our high-level language for modeling Byzantine fault-tolerant distributed systems. Sync is in choreography style [Montesti 2013; Zdancewic et al. 2002], which means a Sync program describes the whole distributed system and can be compiled into programs for individual nodes through *endpoint projection* to be defined later in Section 4.

Sync puts three restrictions on the set of expressible distributed systems. First, Sync does not support branches or loops (except for a meta-level loop over the whole program) due to the problem known as *knowledge of choice* (KoC) [Castagna et al. 2011], where some nodes may not know which branch to take. This becomes sort of a chicken-and-egg situation with failures, as consensus protocols are often used to resolve KoC in a fault-tolerant way in practice. Second, Sync cannot model various cryptographic primitives, so distributed systems that depend on them cannot be expressed. How to formally and modularly reason about distributed systems relying on more sophisticated usage of cryptography is an open problem on its own. We left supporting specific primitives for future work. Last, we enforce a total order on the communication actions in the system. As a result, this version of Sync only supports essentially streamlined programs, whose Async semantics satisfy the definition of *communication-closed* programs [Damian et al. 2019].

At a high level, the denotational semantics of a closed Sync program is a *set* of possible distributed value combinations on some set of nodes. The denotational semantics of a Sync program with free distributed variables is a function that transforms input values for those variables to a set of possible output distributed value combinations.

2.1 Sync Syntax

Our Sync syntax uses the *role* abstraction to describe distributed systems with different kinds of nodes (such as a leader or a replica) while abstracting the number of nodes that will be executing the code associated with a given role. We use L and R as example roles.

A Sync program describes an abstract distributed protocol as a high-level functional program that runs the same code for each node in a given role, albeit on different input values. Its syntax is defined as follows:

Definition 2.1 (Sync Syntax)**.**

Roles	R, L
Meta-terms	t
Channels	c
Distributed Variables	x

Sync Expressions

e	$::=$	x_R	(role-local variables)
		$\lceil t \rceil_R$	(replicated Rocq terms)
		$[t_1, \ldots, t_n]_R$	(vector of Rocq terms)
		$e_1\ e_2$	(application)

Sync Programs

p	$::=$	$\texttt{ret}\ \{R_1 \mapsto e_1, \ldots, R_n \mapsto e_n\}$	(return a record)
		$\texttt{let}\ x := p_1\ \texttt{in}\ p_2$	(sequencing)
		$\texttt{comm}\ c\ e_m\ e_d\ e_f$	(communication)

Sync is realized via a shallow embedding that inherits all of Rocq's terms at the expression level and adds *programs* (p) that operate over a record mapping roles to vectors of values, one vector element for each node in the given role. Critically, the only way for nodes to interact with each other is through communication. Thus, a Sync program p is essentially a sequence of communication steps, where sequencing is accomplished through `let` and terminated with a `ret`. The `ret` operation takes a record of expressions, one for each role, and those expressions are calculated for each node within that role, by mapping the expression's denotation over a vector of environments, one for each node. Thus, the variables bound in a Sync `let` are records mapping roles to vectors of values. In order to ensure that one node cannot access values from another node, `let`-bound variables are restricted within expressions: a given role can only access its vector of the record, and implicitly, a given node can only access its element of the vector. In our Rocq formalism, we use parametric higher-order abstract syntax (PHOAS) [Chlipala 2008] to represent variables and binding and enforce these constraints.

The program $\texttt{comm}\ c\ e_m\ e_d\ e_f$ involves a channel c, a sender role (given by the type of e_m), and a receiver role (given by the types of e_d and e_f). When executed, all of the sender nodes calculate a message e_m and send that message's value to all of the nodes in the receiver role via the channel c. The receiver role's nodes each run a message handler that accepts the incoming messages and combines them with a folding function e_f and a default value e_d. The channel c is used by the sender and the receiver nodes to distinguish messages for different rounds of communication. Our type system enforces that channel names are unique and used in a particular order. At the implementation level, this uniqueness is realized through sequence numbers.

2.2 Typing for Sync Programs

Our typing rules are largely standard but ensure that each expression is situated for a given role and each channel c is used exactly once in a specified order. The typing judgment for expressions has the form $\Gamma \vdash_R e : \tau$ where Γ is a context mapping variables to record types indexed by roles, and is defined as follows:

$$\textsc{Var}\ \frac{\Gamma(x) = \{R_1 : \tau_1, \ldots, R_n : \tau_n\}}{\Gamma \vdash_{R_i} x_{R_i} : \tau_i} \qquad \textsc{Lift}\ \frac{t : \tau}{\Gamma \vdash_R \lceil t \rceil_R : \tau}$$

$$\textsc{Vector}\ \frac{t_1 : \tau \cdots t_n : \tau}{\Gamma \vdash_R [t_1, \ldots, t_n] : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_R e_1 : \tau' \to \tau \qquad \Gamma \vdash_R e_2 : \tau'}{\Gamma \vdash_R e_1\ e_2 : \tau}$$

For variables, we find the record type associated with the variable by the context Γ, and then extract the type associated with the given role. Note that it is not necessary for a given role to be present for all variables, so x_R is only well-formed when x is in $\mathrm{Dom}(\Gamma)$ and R is in $\mathrm{Dom}(\Gamma(x))$. For inherited Rocq terms, we simply ascribe them the type that Rocq gives them, but situated at a given role. Finally, application is typed as expected.

The typing judgment for programs has the form $\Delta; \Gamma \vdash_{\mathcal{R}} p : \{R_i : \tau_i\}$ where Δ is an *ordered* channel context mapping channel names c to a triple (S, R, τ) of a sending role, a receiving role, and a type for messages to be sent on the channel, and where $\mathcal{R}$ is a set of roles that can be used in the program. The judgment is defined with the following rules:

$$\textsc{Ret}\ \frac{\Gamma \vdash_{R_i} e_i : \tau_i \quad R_i \in \mathcal{R}}{\emptyset; \Gamma \vdash_{\mathcal{R}} \texttt{ret}\ \{R_i \mapsto e_i\} : \{R_i : \tau_i\}}$$

$$\textsc{Let}\ \frac{\Delta_1; \Gamma \vdash_{\mathcal{R}} p_1 : \tau_1 \quad \Delta_2; \Gamma[x : \tau_1] \vdash_{\mathcal{R}} p_2 : \tau_2 \quad \mathrm{Dom}(\Delta_1) \cap \mathrm{Dom}(\Delta_2) = \emptyset}{\Delta_1 +\!\!+ \Delta_2; \Gamma \vdash_{\mathcal{R}} \texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2 : \tau_2}$$

$$\textsc{Comm}\ \frac{\Gamma \vdash_{\{S\}} e_m : \tau_m \quad \Gamma \vdash_{\{R\}} e_d : \tau \quad \Gamma \vdash_{\{R\}} e_f : \tau \to \tau_m \to \tau \quad S, R \in \mathcal{R}}{[c : (S, R, \tau_m)]; \Gamma \vdash_{\mathcal{R}} \texttt{comm}\ c\ e_m\ e_d\ e_f : \{R : \tau\}}$$

For `ret`, we simply check that each expression in the returned record is well-typed at the corresponding role. In this case, the channel context is empty since this command does not do any communication. For $\texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2$, we check that p_1 has a (record) type τ_1 and then extend Γ with the assumption that $x : \tau_1$ to check that p_2 has the (record) type τ_2. Note that here, the channel contexts are checked to be disjoint and are appended in order. For $\texttt{comm}\ c\ e_m\ e_d\ e_f$, the channel context has *only* c associated with a sending role S, receiving role R, and message type τ_m. We check that the message expression (e_m) is typed at the sending role S with type τ_m, and that the default (e_d) and combining function (e_f) expressions are typed at the receiving role R as τ and $\tau \to \tau_m \to \tau$ respectively. Finally, the communication step only returns τ values for the nodes of the receiver role, so the resulting record type is the singleton $\{R : \tau\}$.

2.3 Denotational Semantics for Sync

2.3.1 Notation: For any natural number n and type τ, Vec $n\ \tau$ describes a sequence of n elements of type τ. We use $\vec{v}$ to make clear the value is a vector value and write $\vec{v}@i$ for the operation that projects element i from vector $\vec{v}$. The operations map $f\ \vec{v}$, foldl $f\ d\ \vec{v}$, zip $\vec{v}_1\ \vec{v}_2$, unzip $\vec{v}$ are the usual map, fold, zip, and unzip for vectors. The operation const $n\ v$ replicates the value v n times to produce a vector.

For any set or type τ, $\mathcal{P}(\tau)$ denotes the power set of τ. We also use the power set as a monad with the following notations.

$\mathcal{P}(f) : \mathcal{P}(A) \to \mathcal{P}(B)$	lifting function f into its set version by applying f to all elements
$\mathsf{singleton}(x) = \{x\} : \mathcal{P}(\tau)$	the singleton set, also the monadic return
$\gg= : \mathcal{P}(A) \to (A \to \mathcal{P}(B))$	the bind applies the function to all elements and takes the union

For any record e, we write $e@R$ to extract the record field named by R. We also have a shorthand $\Pi : \mathsf{Vec}\ n\ (\mathcal{P}(\tau)) \to \mathcal{P}(\mathsf{Vec}\ n\ \tau)$ that forms a set of vectors from a vector of sets by enumerating all combinations of each element.

2.3.2 Configuration. The denotation of a Sync program is parameterized by a *configuration*, which is defined as follows:

Definition 2.2 (Distributed System Configuration)**.** Given a role set $\mathcal{R}$, a configuration C defines the following for each role $R \in \mathcal{R}$:

- The total number of nodes n_R and an upper bound on the number of faults to tolerate f_R.
- A set of g_R nodes that are correct or follow the protocol until crash $\mathsf{Good}_R = \{r_1, \ldots, r_{g_R}\}$.
- A set of b_R Byzantine nodes Byz_R, $b_R \leq f_R$ and $b_R + g_R = n_R$.

Given a configuration, we define a relation Netwk_R between a vector of messages sent by nodes in Good_R to a list of messages that a receiver node may possibly receive. We use this relation to calculate all of the "bad" things that can happen in the network, such as dropping a message, permuting the order in which messages are received, or injecting Byzantine messages. In our setting, Netwk_R is defined as a predicate (prop) on a vector and list of messages as follows:

$$\mathsf{Netwk}_R \coloneqq \lambda(\vec{v} : \mathsf{Vec}\ g_R\ \tau).\ (\mathsf{add\_any}\ \vec{v}\ b_R) \gg= \mathsf{perm} \gg= (\lambda\vec{v}.\ \mathsf{trunc}\ \vec{v}\ (n_R - f_R))$$

The term $\mathsf{add\_any}\ \vec{v}\ b_R$ produces the set of all vectors that add up to b_R arbitrary values of type τ to the end of $\vec{v}$. The added messages model those sent by a Byzantine node and can take any value. Note that Byzantine nodes may also send multiple messages, but we assume a correct receiver will only process at most one message from each sender node. The term perm produces the set of all permutations of the input vector and models the arbitrary order in which messages may be received. Finally, the term $\mathsf{trunc}\ \vec{v}\ (n_R - f_R)$ produces all prefixes of $\vec{v}$ whose length is at least $n_R - f_R$. This models a receiver node not receiving some of the messages. Due to the asynchronous network, a node cannot distinguish between a message being delayed and a message never sent. So in practice, a receiver only waits for a certain number of messages before continuing. In our case, the maximum number of messages to wait for is $n_R - f_R$ because a correct node always sends its message, and the total number of faulty nodes, either crashing or Byzantine, is bounded by f_R.

This definition is an over-approximation of the possible behaviors we can observe. For instance, this definition allows Byzantine nodes to generate messages with "secrets" that they may not know. Nevertheless, the over-approximation ensures that we cover all of the cases that need to be considered and is yet precise enough that we can still prove useful properties.

2.3.3 Denotational Semantics. We begin by giving a denotation to Sync program types and in particular, we define:

$$[\![\{R_1 : \tau_1, \ldots, R_n : \tau_n\}]\!] = \{R_1 : \mathsf{Vec}\ g_{R_1}\ \tau_1, \ldots, R_n : \mathsf{Vec}\ g_{R_n}\ \tau_n\}$$

That is, *Sync* records are translated to records of vectors, where the lengths of the vectors are determined by the parameters of the configuration. Next, we lift the translation to variable contexts by defining:

$$[\![[x_1 : \tau_1, \ldots, x_n : \tau_n]]\!] = \{x_1 : [\![\tau_1]\!], \ldots, x_n : [\![\tau_n]\!]\}$$

Thus, environments are represented as records mapping variable names to values, where the values are records mapping roles to vectors. Now we can define the denotation of (derivations of) terms as a recursively defined meta-function with type:

$$[\![\Delta; \Gamma \vdash_{\mathcal{R}} p : \tau]\!] : [\![\Gamma]\!] \to \mathcal{P}([\![\tau]\!])$$

which is defined as follows:[1]:

Definition 2.3 (Denotational Semantics of Sync Programs)**.**

$$\begin{aligned}
[\![\texttt{ret}\ \{R_i \mapsto e_i\}]\!] &= \lambda v.\ \mathsf{singleton}\ \{R_i \mapsto [\![e_i]\!]v\} \\
[\![\texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2]\!] &= \lambda v.\ [\![p_1]\!]v \gg\!= (\lambda y.\ [\![p_2]\!](v[x \mapsto y])) \\
[\![\texttt{comm}\ c\ e_m\ e_d\ e_f]\!] &= \lambda v.\ \mathsf{let}\ \texttt{msgs} \coloneqq [\![e_m]\!]v\ \mathsf{in} \\
&\qquad \mathsf{let}\ \texttt{netmsgs} \coloneqq \mathsf{Netwk}_S(\texttt{msgs})\ \mathsf{in} \\
&\qquad \mathsf{let}\ \texttt{pairs} \coloneqq \mathsf{zip}([\![e_f]\!]v)\ ([\![e_d]\!]v)\ \mathsf{in} \\
&\qquad \mathsf{let}\ \mathsf{app} \coloneqq \lambda(f, d).\ \mathcal{P}(\mathsf{foldl}\ f\ d)\ \texttt{netmsgs}\ \mathsf{in} \\
&\qquad \mathcal{P}(\lambda x.\{R \mapsto x\})(\Pi(\mathsf{map}\ \mathsf{app}\ \texttt{pairs}))
\end{aligned}$$

The definition relies upon a denotation for Sync expressions which has type:

$$[\![\Gamma \vdash_R e : \tau]\!] : [\![\Gamma]\!] \to \mathsf{Vec}\, g_R\, \tau$$

and is defined by:

$$\begin{aligned}
[\![x@R]\!] &= \lambda v.\ v@x@R \\
[\![\lceil t \rceil_R]\!] &= \lambda v.\ \mathsf{const}\ g_R\ t \\
[\![[t_1, \ldots, t_n]_R]\!] &= \lambda v.\ [t_1, \ldots, t_n] \quad (n = g_R) \\
[\![e_1\ e_2]\!] &= \lambda v.\ \mathsf{map}\ (\lambda(f, x).f\ x)\ (\mathsf{zip}\ [\![e_1]\!]v\ [\![e_2]\!]v)
\end{aligned}$$

Working backwards, the denotation of a variable simply extracts that variable from the environment ($v@x$), which yields a record of vectors. We then extract the vector of the corresponding role R at which the expression is situated. The denotation of an embedded Rocq term t replicates the term as a vector for each of the good nodes at the given role. The denotation of a vector of Rocq terms is just that vector, provided its length agrees with the number of nodes in the given role. In practice, we only use vector literals in the meta-theory, so this assumption is easy to discharge. The denotation of an application $e_1\ e_2$ is given by calculating the denotation of e_1 and e_2 respectively, which should return a vector of functions and a vector of values, respectively. We zip the two vectors into a vector of pairs, and then map the apply function across the resulting vector.

The denotation of programs is more involved because we return a set of records of vectors. We use the powerset monad to make the definitions a little simpler. The cases for `ret` and `let` straightforwardly translate into the singleton and bind in the powerset monad. For $\texttt{comm}\ c\ e_m\ e_d\ e_f$, we first compute a vector of messages from all of the good senders. We then use Netwk, which can be treated as a function with the powerset monad, to calculate a set of lists of possible messages, which models the effect of the network. Recall that this set includes the original vector of messages, but also added Byzantine messages, permutations, and dropped messages. We wish to take each possible list of messages in `netmsgs` and fold each receiver's combining function and default value over the list of messages. This is accomplished through $\mathsf{map}\ \mathsf{app}\ \texttt{pairs}$ and Π to convert a vector

[1]Formally, the definition is over derivations of the typing judgment, but to simplify the presentation, we present the definition over Sync syntax

of sets of values to a set of vectors of values. Finally, we must place each of these vectors into a record with a field for the receiver role ($\mathcal{P}(\lambda x.\{R \mapsto x\})$).

It is easy to see that the denotation respects monadic laws, so for instance, lets can be flattened:

$$[\![\texttt{let}\ x_1 \coloneqq (\texttt{let}\ x_2 \coloneqq p_2\ \texttt{in}\ p_3)\ \texttt{in}\ p_4]\!] = [\![\texttt{let}\ x_2 \coloneqq p_2\ \texttt{in let}\ x_1 \coloneqq p_3\ \texttt{in}\ p_4]\!]$$

In fact, every program is equivalent to one in a "let-comm" normal form:

$$p \ ::= \ \texttt{ret}\ \{R_i \mapsto e_i\} \mid \texttt{let}\ x \coloneqq \texttt{comm}\ c\ e_m\ e_d\ e_f\ \texttt{in}\ p$$

which we will leverage in the proof.

2.3.4 An Example Program. We go through the `SimpleVote` program in Figure 1a to provide more intuition. The `SimpleVote` program is typed by

$$[c : (R, L, \mathbb{B})]; [p : \{L : \mathbb{B}\}, x : \{R : \mathbb{B}\}] \vdash_{\{L,R\}} \texttt{SimpleVote} : \{L : \texttt{option}\ \mathbb{B}\}$$

It requires a single Boolean value as input at all good nodes and produces an optional Boolean value at the leader as the output.

For demonstration, we use a concrete configuration where there is a single leader node and $n_R = 4$ replica nodes, r_1, r_2, r_3, r_B, of which r_B is Byzantine and $f_R = b_R = 1$.

The denotation of `SimpleVote` is thus a function of type

$$\{p : \{L : \text{Vec}\ 1\ \mathbb{B}\}, x : \{R : \text{Vec}\ 3\ \mathbb{B}\}\} \to \mathcal{P}(\{L : \text{Vec}\ 1\ (\texttt{option}\ \mathbb{B})\})$$

We use $\top$ and $\bot$ to represent Boolean true and false. Suppose the concrete input is $\{p \mapsto \{L \mapsto [\top]\}, x \mapsto \{R \mapsto [\top, \top, \bot]\}\}$, which means the leader's input is $\top$ and the replicas' inputs are $[\top, \top, \bot]$ for r_1, r_2, and r_3 respectively. All possible lists of messages that the leader node could receive are given by the Netwk_R relation. Let $\top(r_i)$ represent a value $\top$ sent by node r_i. The exact set is all permutations of

$$\{\top(r_1), \top(r_2), \top(r_B)\}, \{\top(r_1), \bot(r_3), \top(r_B)\}, \{\top(r_2), \bot(r_3), \top(r_B)\},$$
$$\{\top(r_1), \top(r_2), \bot(r_B)\}, \{\top(r_1), \bot(r_3), \bot(r_B)\}, \{\top(r_2), \bot(r_3), \bot(r_B)\},$$
$$\{\top(r_1), \top(r_2), \bot(r_3), \top(r_B)\}, \{\top(r_1), \top(r_2), \bot(r_3), \bot(r_B)\}, \{\top(r_1), \top(r_2), \bot(r_3)\}$$

Because the input to the leader node is $\top$, `fcnteq` counts the number of $\top$ elements in the received messages, and the order of the elements does not influence its output. Thus, the set of possible communication results is $\{1, 2, 3\}$. `calc_dec` outputs Some $\top$ if there are at least $2 = (n - 2 * f)$ votes for $\top$ and None otherwise. So, the set of possible outputs of the leader node is {Some $\top$, None} given this particular input. This leads to the set of possible final outputs $\{\{L \mapsto [\text{Some}\ \top]\}, \{L \mapsto [\text{None}]\}\}$.

3 Case Studies

In this section, we demonstrate the use of the denotational semantics of Sync for proving safety properties of two concrete consensus protocols in a Hoare-like style. We sketch the steps we followed in our mechanized proof, which is about 2.5k LoC in total, including all the definitions and proofs of the languages and the case studies.

3.1 Modeling Non-terminating Consensus Protocols

In general, the goal of a consensus protocol is to have a collection of nodes agree on a common value, even in the presence of faulty nodes and unpredictable network behaviors. The famous FLP impossibility result [Fischer et al. 1985] states that in a fully asynchronous network, there is no non-trivial consensus protocol that is safe, guaranteed to terminate, and can tolerate even a single

crash failure. Because of this, consensus protocols are usually not guaranteed to terminate without making stronger assumptions.

In our framework, we model protocols that execute a Sync program in a loop. Since we only consider safety properties, it is sufficient to consider the finite unwindings of the loop. We assume the loop body is a Sync program that takes in some input and produces an output that contains the input to the next iteration. We use the construct b^k to unroll the loop k times and provide fresh channel names for all iterations.

More formally, a protocol body b is of the form $\nu c_1, \ldots, c_n.\lambda x.p$ where the free channel names in p are bound via the fresh quantifier ν and x is the input to the protocol. Given two protocol bodies b_1 and b_2, their concatenation $b_1 +\!\!+ b_2$ is defined as follows:

$$(\nu c_1, \ldots, c_n.\lambda x.p) +\!\!+ (\nu c'_1, \ldots, c'_n.\lambda x'.p') = \nu c_1, \ldots, c_n, c'_1, \ldots, c'_n.\lambda x.p[\lambda x'.p']$$

where $p[\lambda x'.p']$ is a form of monadic substitution defined by:

$$\begin{array}{rcl} (\texttt{ret}\ \{R_i \mapsto e_i\})[\lambda x'.p'] & = & p'[e_i/x'@R_i] \\ (\texttt{let}\ x \coloneqq p_1\ \texttt{in}\ p_2)[\lambda x'.p'] & = & \texttt{let}\ x \coloneqq p_1\ \texttt{in}\ (p_2[\lambda x'.p']) \\ (\texttt{comm}\ c\ e_m\ e_d\ e_f)[\lambda x'.p'] & = & \texttt{let}\ x' \coloneqq \texttt{comm}\ c\ e_m\ e_d\ e_f\ \texttt{in}\ p' \end{array}$$

Note that in the `ret` case, our syntax does not support substituting the record expression for the variable x, so we must project each role R's expression and substitute that for each occurrence of x_R. With this definition in hand, given a protocol body b, we can define b^k as:

$$\begin{array}{rcl} b^0 & = & b \\ b^{k+1} & = & b +\!\!+ b^k \end{array}$$

3.2 Bosco

Bosco [Song and van Renesse 2008] stands for **B**yzantine **O**ne-**S**tep **CO**nsensus, a consensus algorithm that only performs a single round of communication per iteration. It is a symmetric protocol with only one role, which we denote as R for replica. We assume two global parameters: n is the total number of nodes, and f is the number of Byzantine faults to tolerate. For simplicity, we analyze the binary version of the protocol, where the goal is to have all nodes decide on either $\top$ or $\bot$. A single iteration of Bosco is defined below:

Definition 3.1 (Binary Bosco in Sync Syntax)**.**

```
1 Bosco (v : {R : 𝔹}) : {R : (option 𝔹) * 𝔹} ≔
2   let cnts ≔ comm c v_R ⌈(0, 0)⌉_R ⌈fcntb⌉_R in
3   ret (⌈mkdec⌉_R cnts_R)
  where
4 fcntb ≔ λ (cnt_⊤, cnt_⊥), v. if v then (cnt_⊤+1, cnt_⊥) else (cnt_⊤, cnt_⊥+1)
5 mkdec ≔ λ (cnt_⊤, cnt_⊥).
6   let (newv, cnt) ≔ if cnt_⊤ ≥ cnt_⊥ then (⊤, cnt_⊤) else (⊥, cnt_⊥) in
7   if cnt * 2 > n + 3 * f then (Some newv, newv) else (None, newv)
```

An iteration of the Bosco protocol requires a single Boolean value at every node as input. On line 2, every node broadcasts its input value v_R to every other node and counts the number of received $\top$ and $\bot$. On line 3, each node computes the decision and the input to the next iteration with function `mkdec`. On line 6, in `mkdec`, a node compares the numbers of $\top$s and $\bot$s it receives and records the value with more votes as the `newv` and the number of its occurrences as `cnt`. On

line 7, each node produces a decision value dec, which is Some newv if cnt is greater than the threshold $\frac{n+3f}{2}$ or None if otherwise. newv is to be passed to the next iteration as the input.

Using our semantic definitions, we prove the following two properties for Bosco:

(1) (One Step): When $n > 7f$, if all correct nodes have the same input B in some iteration, then all correct nodes decide and output Some B in that iteration.
(2) (Agreement): When $n > 3f$, if a correct node outputs Some B_1 in an iteration and another correct node, not necessarily a different one, outputs Some B_2 in a not necessarily different iteration, then $B_1 = B_2$.

Definition 3.2 (Bosco Configuration)**.** We assume a configuration C with one role R, good nodes $\text{Good}_R = \{r_1, \ldots, r_{n-b}\}$, and b Byzantine nodes where $b \leq f$.

Proving the one-step property involves a single iteration, and the property is defined formally as:

Theorem 3.1 (Strongly one-step)**.** If $n > 7f$, then for all Boolean values B, input vectors $\vec{x}$ such that $\forall r_i, \vec{x}@r_i = B$, and output values $\{R \mapsto \vec{y}\} \in [\![P]\!]\{x \mapsto \{R \mapsto \vec{x}\}\}$, we have $\forall r_i, \vec{y}@r_i = (\text{Some } B, B)$.

Proof. We first give a lemma about Netwk_R. For convenience, we define $\#_v(\ell)$ as the number of occurrences of v in the list ℓ.

$$\#_v(\ell) \coloneqq \text{foldl } (\texttt{fcnteq}\ v)\ 0\ \ell \text{ where } \texttt{fcnteq} \coloneqq \lambda v, c, v'.\ \text{if } v == v' \text{ then } c + 1 \text{ else } c$$

Lemma 3.2. For any role R and any ℓ such that $|\ell| = n_R - b_R$, we have:

$$\forall \ell' \in \text{Netwk}_R(\ell), v, \#_v(\ell) - f_R \leq \#_v(\ell') \leq \#_v(\ell) + b_R$$

This lemma can be proven by following the definition of Netwk_R and $\#_v(\ell)$.

Our main proof follows the structure of the Bosco program's semantics. We start with the precondition $\forall r_i, \vec{x}@r_i = B$. By definition, line 2 applied to $\vec{x}$ reduces to:

$$\Pi(\text{map } (\lambda(f, d).\ \mathcal{P}(\text{foldl } f\ d)\ \text{Netwk}_R(\vec{x}))\ (\texttt{const}\ (n - b)\ (\texttt{fcntb}\ (0, 0)))$$

So for any output of line 2, cnts, we have

$$\forall r_i, \texttt{cnts}_R@r_i \in \mathcal{P}(\text{foldl } \texttt{fcntb}\ (0, 0))\ \text{Netwk}_R(\vec{x})$$

By the definition of the power set monad, $\mathcal{P}(\text{foldl } \texttt{fcntb}\ (0, 0))$ applies the function to every element of the input. Thus,

$$\exists \ell \in \text{Netwk}_R(\vec{x}), \texttt{cnts}_R@r_i = \text{foldl } \texttt{fcntb}\ (0, 0)\ \ell$$

By definition, $\text{foldl } \texttt{fcntb}\ (0, 0)\ \ell = (\#_\top(\ell), \#_\bot(\ell))$. Let $(\texttt{cnt}_\top, \texttt{cnt}_\bot) \coloneqq \texttt{cnts}_R@r_i$. Because of $\forall r_i, \vec{x}@r_i = B$, $\#_B(\vec{x}) = n - b$ and $\#_{\neg B}(\vec{x}) = 0$. By the lemma above, we know:

$$\texttt{cnt}_B = \#_B(\ell) \geq \#_B(\vec{x}) - f = n - b - f$$
$$\texttt{cnt}_{\neg B} = \#_{\neg B}(\ell) \leq \#_{\neg B}(\vec{x}) + b = b$$

For line 3, by the definition of mkdec, because $n > 7f$, we have:

$$\texttt{cnt}_B \geq n - b - f \geq n - 2f > 5f \geq b \geq \texttt{cnt}_{\neg B}$$

So, $\texttt{newv} = B$ and $\texttt{cnt} = \texttt{cnt}_B \geq n - 2f > \frac{n+3f}{2}$, which leads to the final output $(\text{Some } B, B)$. □

To formalize the agreement property, we first need some auxiliary definitions:

$$\text{Step}^k(\vec{x}) \coloneqq ([\![\texttt{Bosco}^k]\!] \circ \mathcal{P}(@R) \circ \mathcal{P}(\texttt{unzip}))\{x \mapsto \{R \mapsto \vec{x}\}\}$$
$$\text{Decide}_B(\vec{y}) \coloneqq \exists r_i.\vec{y}@r_i = \text{Some } B$$
$$\text{Comply}_B(\vec{y}) \coloneqq \forall r_i.\vec{y}@r_i = \text{Some } B \vee \vec{y}@r_i = \text{None}$$

Step^k runs the protocol for $k+1$ steps and extracts the results from the record. Decide_B is a predicate on a vector that holds when there is an element equal to Some B, which means a node has decided B, and Comply_B is a predicate on a vector that holds when every element is Some B or None, which means each node either decides B or nothing.

We prove a stronger result that implies the agreement property.

Lemma 3.3 (Agreement'). If $n > 3f$, then

$$\forall B, \vec{x}, (\vec{y}, \vec{z}) \in \mathsf{Step}^0(\vec{x}), \mathsf{Decide}_B(\vec{y}) \\ \Rightarrow [\mathsf{Comply}_B(\vec{y}) \wedge \forall k, (\vec{y}_k, \_) \in \mathsf{Step}^k(\vec{z}), \mathsf{Comply}_B(\vec{y}_k)]$$

Proof. Proofs like these revolve around some *univalent condition*. It is a predicate on the system state. All reachable states from a state that satisfies the predicate can only decide on one certain value. In our case, thanks to our synchronous functional semantics, the exact univalent condition is exactly the weakest precondition such that the newv computed on line 6 can only be a certain value for all the possible sets of messages being received, as follows:

$$UC'_B(\vec{x}) \coloneqq \forall \vec{x'} \in \mathsf{Netwk}_R(\vec{x}), \mathsf{snd}(\mathtt{mkdec}(\mathtt{foldl}\ \mathtt{fcntb}\ (0,0)\ \vec{x'})) = B$$

Due to the asymmetrical nature of the $\leq$ comparison, this is equivalent to:

$$UC'_\top(\vec{x}) = \#_\top(\vec{x}) \geq \frac{n+f}{2} \quad UC'_\bot(\vec{x}) = \#_\bot(\vec{x}) > \frac{n+f}{2}$$

The decision procedure for Bosco does not utilize this asymmetry [Song and van Renesse 2008]. An optimization is possible by using $\geq$ instead of $>$ in the condition on line 7 only when newv is $\top$. Here, for simplicity, we use a slightly stronger symmetric predicate that is still sufficient to prove our result:

$$UC_B(\vec{x}) \coloneqq \#_B(\vec{x}) > \frac{n+f}{2}$$

And we prove the agreement property by proving the following:

$$\forall \vec{x}, (\vec{y}, \vec{z}) \in \mathsf{Step}^0(\vec{x}), \mathsf{Decide}_B(\vec{y}) \Rightarrow UC_B(\vec{x}) \tag{1}$$

$$\forall \vec{x}, k, (\vec{y}, \vec{z}) \in \mathsf{Step}^k(\vec{x}), UC_B(\vec{x}) \Rightarrow [\mathsf{Comply}_B(\vec{y}) \wedge UC_B(\vec{z})] \tag{2}$$

We prove (1) by walking through the program backward with $\mathsf{Decide}_B(\vec{y})$. Because our program is nondeterministic, walking through the program backward requires us to infer a predicate that covers all possible input values that can lead to any output values satisfying the postcondition.

- On line 3, we know that for some r_i, dec = Some B.
- By the definition of mkdec, it is necessary to have $\mathtt{cnt}_B > \frac{n+3f}{2}$.
- On line 2, by Lemma 3.2, $\#_B(\vec{x}) \geq \mathtt{cnt}_B - b > \frac{n+f}{2}$, which is our goal, UC_B.

For (2), we do an induction on k. The only case that needs to be proven is $k = 0$. We walk through the program forward to infer a predicate that covers all possible outputs.

- On line 2, by Lemma 3.2 and $UC_B(\vec{x})$, for any r_i and any possible network, we have $\mathtt{cnt}_B > \frac{n+f}{2} - f \wedge \mathtt{cnt}_{\neg B} \leq \#_{\neg B}(\vec{x}) + b < n + f - \frac{n+f}{2} = \frac{n+f}{2}$. So $\mathtt{cnt}_B > \mathtt{cnt}_{\neg B}$.
- By the definition of mkdec, newv = B and dec = Some newv = Some $B \vee$ dec = None.
- On line 3, we have $\forall r_i,\ (\vec{y}@r_i = \text{Some } B \vee \vec{y}@r_i = \text{None}) \wedge (\vec{z}@r_i = B)$.

$\forall r_i,\ \vec{y}@r_i$Some $B \vee \vec{y}@r_i$ = None implies $\mathsf{Comply}_B(\vec{y})$. $\forall r_i,\ \vec{z}@r_i = B$ implies $\#_B(\vec{z}) = n - b \geq n - f$. Because $n > 3f$, we have $\#_B(\vec{z}) > \frac{n+f}{2}$, which is UC_B. □

3.3 Sequential Paxos

Sequential Paxos, or SeqPaxos for short, is a variant of the Synod consensus protocol used by Paxos [Lamport 1998]. Similar to the original single-decree Paxos protocol, it is a crash fault-tolerant protocol with a single leader node and an arbitrary number of so-called acceptor nodes, which we will call *replicas*.

Let n be the total number of replicas and f be the number of replica crashes to tolerate. The single leader may also crash. We assume the domain of the consensus value is of type $\mathbb{V}$, for which a decidable equality exists. Additionally, we assume that there is a default value for the leader to propose in each iteration, denoted as a global variable $\texttt{default} : \mathbb{N} \to \mathbb{V}$. A single iteration of SeqPaxos defined in Sync is listed below:

Definition 3.3. SeqPaxos

```
1  SeqPaxos (x : {L : ℕ, R : (option 𝕍) * ℕ})
         : {L : (option 𝕍) * ℕ, R : (option 𝕍) * ℕ} :=
2    let maxv := comm c1 xR ⌈(None, 0)⌉L ⌈fmaxr⌉L in
3    let p := ret {L ↦ ⌈pickp⌉L maxvL (⌈default⌉L (xL))} in
4    let y := comm c2 (⌈pair⌉L pL xL) xR ⌈update⌉R in
5    let cnt := comm c3 yR ⌈0⌉L (⌈fcnteq⌉L xL) in
6    ret {L ↦ ⌈pair⌉L (⌈mkdec⌉L cntL pL) (⌈add⌉L xL ⌈1⌉L), R ↦ yR}
   where
7    fmaxr := λ (v, r), (v', r'). if r < r' then (v', r') else (v, r)
8    pickp := λ (ov, _), d. match ov with | Some v ⇒ v | None ⇒ d end
9    update := λ _, (v, r). (Some v, r)
10   fcnteq := λ r, c, (_, r'). if r == r' then c + 1 else c
11   mkdec := λ c, p. if c > f then Some p else None
```

In the SeqPaxos protocol, the leader takes a single natural number, which is the current round number. The replicas take a pair of an optional $\mathbb{V}$ value and a natural number. The number is the latest round number ever received, and the value is the proposal of that round. To initiate the protocol, the leader should be given the input 1 and the replicas the dummy values None and 0.

SeqPaxos performs three rounds of communication in each iteration. In the first round of communication on line 2, all the replicas send their local value and round number to the leader. The leader then finds the largest round number and the proposal associated with the round number. On line 3, the leader computes the proposal of the iteration. If the proposal associated with the largest round number received is Some v, the proposal is v. Otherwise, the proposal is set to the default value of that iteration. On line 4, the leader broadcasts the proposal with its round number to all the replicas. For each replica, if it receives the message from the leader, it updates its local value and round number to the new local value and round number; otherwise, it keeps its old local value and round number. On line 5, all the replicas send their local value and round number to the leader again. This time, the leader counts how many replicas have updated their local value and round number to the latest proposal and round number pair. On line 6, if the number of up-to-date replicas is greater than f, the leader decides their proposal as the decision of the iteration; otherwise, no decision is made. The leader returns a pair of the decision and an incremented round number while each replica returns its latest value.

Definition 3.4 (SeqPaxos Configurations)**.** SeqPaxos requires a configuration C to have two roles, L and R. L is the leader role, $\text{Good}_L = \{l\}$, $b_L = 0$. We set $f_L = 1$ to model that the leader may crash. R is the replica role, $\text{Good}_R = \{r_1, \ldots, r_n\}$. We also set $b_R = 0$ and $f_R = f$ to model the assumption that up to f replicas may crash.

While we do not directly model the crash behavior, we claim the above definition covers all possible cases for SeqPaxos. This is because we are only reasoning about finite iterations, and there is no difference between a node that has crashed at some point and a node that has certain messages sent from and to it dropped by Netwk.

The special initial input to SeqPaxos is defined as $\texttt{init} = \{L \mapsto [1]; R \mapsto \vec{x}\}$ where $\vec{x}$ is n copies of $(\text{None}, 0)$.

We formally state and prove the agreement property for SeqPaxos.

Theorem 3.4 (Agreement)**.** If $n > 2f$,

$$\begin{aligned}&\forall D, i, \{L \mapsto [(d_i, r_i)]; R \mapsto \vec{x}_i\} \in [\![\mathsf{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\}),\ \ d_i = \text{Some } D \Rightarrow \\ &\quad \forall j > i, \{L \mapsto [(d_j, \_)]; R \mapsto \_\} \in [\![\mathsf{SeqPaxos}^{j-i-1}]\!](\{x \mapsto \{L \mapsto [r_i]; R \mapsto \vec{x}_i\}\}), \\ &\quad d_j = \text{Some } D \vee d_j = \text{None}\end{aligned}$$

Proof. We have the following lemma about the inputs to any iteration i, which can be proven by induction on i and stepping through the program.

Lemma 3.5. $\forall i, \{L \mapsto [(\_, r)]; R \mapsto \vec{x}\} \in [\![\mathsf{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\})$, implies:

- $r = i + 2 \wedge$
- $\forall u, \mathsf{snd}(\vec{x}@u) < i + 2 \wedge$
- $\forall u, 0 < \mathsf{snd}(\vec{x}@u) \Rightarrow \exists v, \mathsf{fst}(\vec{x}@u) = \text{Some } v \wedge$
- $\forall u, w, \mathsf{snd}(\vec{x}@u) = \mathsf{snd}(\vec{x}@w) \Rightarrow \vec{x}@u = \vec{x}@w.$

We again define the univalent condition UC_D for SeqPaxos as the weakest precondition[2] of line 3 always resulting in value D.

$$UC_D(\vec{x}) \coloneqq \forall \ell \in \mathsf{Netwk}_R(\vec{x}), \mathsf{fst}(\mathsf{foldl}\ \texttt{fmaxr}\ (\text{None}, 0)\ \ell) = \text{Some } D$$

For the agreement property, we prove the following:

$$\begin{aligned}&\forall D, i, \{L \mapsto [(d_i, \_)]; R \mapsto \vec{x}_i\} \in [\![\mathsf{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\}), \\ &\quad d_i = \text{Some } D \Rightarrow UC_D(\vec{x}_i) \qquad (1) \\ &\forall D, i, \{L \mapsto \_; R \mapsto \vec{x}_i\} \in [\![\mathsf{SeqPaxos}^i]\!](\{x \mapsto \texttt{init}\}), UC_D(\vec{x}_i) \Rightarrow \\ &\quad \forall j > i, \{L \mapsto [(d_j, \_)]; R \mapsto \vec{x}_j\} \in [\![\mathsf{SeqPaxos}^{j-i-1}]\!](\{x \mapsto \{L \mapsto [r_i]; R \mapsto \vec{x}_i\}\}), \\ &\quad\quad (d_j = \text{Some } D \vee d_j = \text{None}) \wedge UC_D(\vec{x}_j) \qquad (2)\end{aligned}$$

It is worth noting that our UC predicate is weaker than the strongest postcondition of lines 5 and 6 deciding D, which means the protocol is actually safe to decide in more cases. An optimization for the original Paxos protocol that utilizes this is described in [Goldweber et al. 2020]. Here, we prove the property with this precise condition, allowing optimizations that weaken the decision condition to be verified compositionally and reuse the proof of (2).

To prove (1), we work backward to find all possible intermediate cases that may lead to $d_i = \text{Some } D$ and then prove they entail $UC_D(\vec{x}_i)$.

- By Lemma 3.5, the round number input for the round that produces $\vec{x}_i$ is $i + 1$.

[2]We treat `default` as some value that cannot be used outside of the particular round at runtime.

- By the definition of mkdec on line 11, we know on line 6, for the single leader node l, $\mathsf{cnt} > f \wedge p = D$.
- On line 5, we have

$$\exists \ell \in \mathsf{Netwk}_R(\vec{x}_i), \mathsf{foldl}\ (\mathtt{fcnteq}\ (i+1))\ 0\ \ell > f$$

- By line 4, Lemma 3.2, and $b_R = 0$, $\#_{(\mathsf{Some}\ D, i+1)}(\vec{x}_i) > f - b_R = f$.
- Unfolding the definition of UC_D, we need to prove:

$$\forall \ell \in \mathsf{Netwk}_R(\vec{x}_i), \mathsf{fst}(\mathsf{foldl}\ \mathtt{fmaxr}\ (\mathsf{None}, 0)\ \ell) = \mathsf{Some}\ D$$

  By Lemma 3.2 again, $\#_{(\mathsf{Some}\ D, i+1)}(\ell) > f - f = 0$. So there exists w, $w \in \ell = (\mathsf{Some}\ D, i+1)$. By Lemma 3.5,

$$\forall u, \mathsf{snd}(\vec{x}_i@u) \leq i+1 \wedge \mathsf{snd}(\vec{x}_i@u) = i+1 \Rightarrow \vec{x}_i@u = \vec{x}_i@w = (\mathsf{Some}\ D, i+1)$$

  which means $i+1$ is the largest round number, and every replica with round number $i+1$ has value $(\mathsf{Some}\ D, i+1)$. So by the definition of fmaxr,

$$\mathsf{foldl}\ \mathtt{fmaxr}\ (\mathsf{None}, 0)\ \ell = (\mathsf{Some}\ D, i+1)$$

To prove (2), we perform induction on j. It suffices to prove for a single iteration, and we do a forward pass through the program.

- By Lemma 3.5, the round number output for the round that produces $\vec{x}_i$ is $i+2$.
- For lines 2 and 3, by the definition of UC_D, we have $p = D$. This implies $\mathsf{dec} = \mathsf{Some}\ D \vee \mathsf{dec} = \mathsf{None}$ on line 6.
- On line 4, our goal is to prove UC_D holds on the new values of the replicas, $y@R$. By the definition of update and Lemma 3.5,

$$\forall u, y@R@u = (\mathsf{Some}\ D, i+2) \vee y@R@u = \vec{x}_i@u$$

  So each node nondeterministically picks between switching to the new value $(\mathsf{Some}\ D, i+2)$ and keeping the old value $\vec{x}_i@u$. This property is preserved by the Netwk relation, more formally:

$$\forall \ell \in \mathsf{Netwk}_R(y@R), \exists \ell' \in \mathsf{Netwk}_R(\vec{x}_i), \forall u, \ell@u = (\mathsf{Some}\ D, i) \vee \ell@u = \ell'@u$$

  For any $\ell \in \mathsf{Netwk}_R(y@R)$, there are two cases.
- If $(\mathsf{Some}\ D, i+2) \in \ell$, then $\mathsf{foldl}\ \mathtt{fmaxr}\ (\mathsf{None}, 0)\ \ell = (\mathsf{Some}\ D, i+2)$.
- Otherwise, all nodes whose messages were received did not update their local values. Thus, $\ell = \ell'$ and $\ell \in \mathsf{Netwk}_R(\vec{x}_i)$. Because of $UC_D(\vec{x}_i)$, $\mathsf{fst}(\mathsf{foldl}\ \mathtt{fmaxr}\ (\mathsf{None}, 0)\ \ell) = \mathsf{Some}\ D$. Thus, $UC_D(\vec{x}_{i+1} = y@R)$.

□

4 Low-Level Semantics

In this section, we define Async, whose semantics is a labeled transition system that serves as a low-level operational semantics for general distributed systems, and that serves as a target language for compiling Sync programs. Async provides a formal basis for proving the adequacy of the denotational reasoning we propose for Sync programs.

We first clarify the network and adversary assumptions used Async's semantics is modeled after and then construct the semantics by composing multiple instances of local labeled transition systems, Async nodes and Async channels. An Async node models the behaviors of an individual non-Byzantine node in the system, while an Async channel models the influence of the network and Byzantine nodes in a single logical round of communication.

4.1 Network and Adversary Assumptions

We assume any network message contains two parts of information: some message payload x, which is a well-typed value, and an auxiliary header, which includes information such as an identifier for the channel this message is for, the sender's identity, the receiver's identity, *etc.* The header information is invisible to the distributed program and therefore not modeled in the semantics; it is used only by the runtime system to provide specific guarantees.

We assume the network does not duplicate messages, which can be enforced by suppressing the duplicates at the receiving end. We also require the messages to be *authenticated*: if a message with the payload x is received by some non-Byzantine receiver p, the runtime of p can check that the identity of the sender is some node q in the system using the information in the header. Furthermore, if q is non-Byzantine, q must have sent the message with payload x. Cryptographic signatures are a common mechanism used to satisfy these assumptions, but we do not model signatures directly.

For the delivery of messages, we assume the network is asynchronous and reliable. Asynchronous means that there is no bound on the delay between the sender node sending a message and the receiver node receiving and processing this message. Reliable means that a message sent from a correct sender to a correct receiver will eventually arrive, which is necessary for the system to make progress. This eventual arrival can be achieved by resending the messages.

We assume a powerful adversary that overestimates the capabilities of a realistic one. In particular, our adversary has complete control over the set of faulty nodes, has sufficient computational resources, is capable of sending out arbitrary messages, and controls the message delivery schedule.

However, we do not explicitly model the adversary doing any of the following:

- Attacking the liveness of the network, such as in denial-of-service attacks.
- Modifying messages sent by non-Byzantine nodes as in man-in-the-middle attacks. Our authenticated network assumption excludes the cases where the adversary forges messages that appear to have been sent by a correct node.
- Sending ill-formed messages, such as incomplete auxiliary information or an ill-typed value. Although a common source of vulnerabilities in reality, this kind of messages can be filtered by a correct parser and dropped. In our case, we do not model parsing nor verify its correctness.
- Sending multiple messages with the same header but different payloads. We assume the receiver simply drops all but one message from a given sender.

Under our assumptions, the Byzantine adversary can be modeled as a stateless entity because any series of events that can happen during the execution does not change its capabilities. Furthermore, because the only ways the adversary can affect the execution of correct nodes are by sending well-formed messages that contain arbitrary contents and manipulating the delivery schedule, we model the adversary's influence through a special action of the network that creates well-formed messages out of thin air and consider all possible schedules of the network.

4.2 Async Node Semantics

Recall that a labeled transition system $\langle S, \Lambda, \rightarrow \rangle$ includes a set of states S, a set of labels Λ, and a transition relation $\rightarrow$ over $S \times \Lambda \times S$ that relates a starting state, a label, and a next state. A behavior or *trace* of a labeled transition system from a specific initial state is a sequence of labels and states where each element is permitted by $\rightarrow$ from the previous state, given the label. We write $s \xrightarrow{\ell} t$ for when $(s, \ell, t) \in \rightarrow$. If L is a sequence of labels, we write $s \xrightarrow{L} t$ to mean if L is empty, then $t = s$, and if $L = \ell :: L'$, then there exists a t' such that $s \xrightarrow{\ell} t'$ and $t' \xrightarrow{L'} t$.

The core of an Async node is a program represented by the following definition in Rocq, which we will use to represent states in our transition system for nodes.

Definition 4.1 (Async Monad for Async Node Syntax)**.**

```
Inductive Async(T : Type) : Type :=
  | return : T → Async T
  | sendThen : ∀c : Channel, msg_t(c) → Async T → Async T
  | rcvThen : ∀c : Channel, (list(msg_t c) → Async T) → Async T.

Fixpoint bind{T U}(e : Async T)(f : T → Async U) :=
  match e with
  | return v ⇒ f v
  | sendThen c m k ⇒ sendThen c m (bind k f)
  | rcvThen c g ⇒ rcvThen c (λ m. bind (g m) f)
  end.

Definition send c m := sendThen c m (return tt).
Definition receive c (d : T) (f : T → msg_t c → T) := rcvThen c (λm. return (foldl f d m)).
```

Async T is a recursively defined type, which describes three kinds of computations: returning a value of type T; sending a message to channel c, and then continuing with another computation; receiving from a channel c a list of messages, which are fed to a function to produce a continuation.

Some auxiliary operations will be needed to build Async node programs below. The operation bind t f "appends" f onto the leaves of Async node program t. More properly, the leaves of t must be of the form return e, and the bind replaces these leaves with f e. The definitions for send and receive build corresponding Async node program trees with trivial continuations that immediately return. Note that bind and return form a monad.

To give meaning to Async node programs, we formulate a labeled transition system $\langle S_t, \Lambda_t, \rightarrow_t \rangle$:

Definition 4.2 (Async Node Semantics)**.** The state, S_t is an Async node program of some return type T.

Λ_t has two kinds of labels:

$$l_t ::= \text{send } c\ v \mid \text{receive } c\ \vec{v}$$

The transition relation is defined by:

$$\text{(SEND)}\ \text{sendThen } c\ v\ k \xrightarrow{\text{send } c\ v} k \qquad\qquad \text{(RECEIVE)}\ \text{rcvThen } c\ f \xrightarrow{\text{receive } c\ \vec{v}} f(\vec{v})$$

To obtain the initial state for a given node, we now define a translation from Sync programs and roles to Async node programs, which is a form of *endpoint projection* because Sync is in choreography style.

Definition 4.3 (Compiling Sync to Async Node Programs)**.** Given a record type $\tau = \{R_1 : \tau_1, \ldots, R_n : \tau_n\}$ we define $(\!|\, \tau \,|\!)_{R_i} = \tau_i$. When R is not a field of τ, we take $(\!|\, \tau \,|\!)_R$ to be unit. Given a variable context $\Gamma = [x_1 : \tau_1, \ldots, x_n : \tau_n]$ we define $(\!|\, \Gamma \,|\!)_R$ to be $[x_1 : (\!|\, \tau_1 \,|\!)_R, \ldots, x_n : (\!|\, \tau_n \,|\!)_R]$. Then, given a Sync program p such that $\Delta; \Gamma \vdash_{\mathcal{R}} p : \tau$ and a role $R \in \mathcal{R}$ and node i in role R, the compilation of p at role R

and node i denoted by $(\!| p |\!)_{R,i}$ yields a term t such that $(\!| \Gamma |\!)_R \vdash t : \mathsf{Async}(\!| \tau |\!)_R$ and is defined by:

$$(\!| \mathtt{ret}\ \{R_j \mapsto e_j\} |\!)_{R,i} = \begin{cases} \mathsf{return}\ (\!| e_j |\!)_{R,i} & R = R_j \\ \mathsf{return}\ \mathtt{tt} & \forall j, R \neq R_j \end{cases}$$

$$(\!| \mathtt{let}\ x \coloneqq p_1\ \mathtt{in}\ p_2 |\!)_{R,i} = (\!| p_1 |\!)_{R,i} \gg\!= \lambda x.(\!| p_2 |\!)_{R,i}$$

$$(\!| \mathtt{comm}\ c\ e_m\ e_d\ e_f |\!)_{R,i} = \begin{cases} \mathsf{let}\ \_ \leftarrow \mathsf{send}\ c\ (\!| e_m |\!)_{R,i}\ \mathsf{in} & \\ \quad \mathsf{receive}\ c\ (\!| e_d |\!)_{R,i}\ (\!| e_f |\!)_{R,i} & \backslash\backslash\ R \text{ sending and receiving} \\ \mathsf{send}\ c\ (\!| e_m |\!)_{R,i} & \backslash\backslash\ R \text{ sending but not receiving} \\ \mathsf{receive}\ c\ (\!| e_d |\!)_{R,i}\ (\!| e_f |\!)_{R,i} & \backslash\backslash\ R \text{ receiving but not sending} \\ \mathsf{return}\ \mathtt{tt} & \backslash\backslash\ R \text{ not sending nor receiving} \end{cases}$$

The definition assumes a translation for expressions $(\!| e |\!)_{R,i}$ which simply removes the R from variables and terms:

$$\begin{aligned} (\!| x_R |\!)_{R,i} &= x \\ (\!| \lceil t \rceil_R |\!)_{R,i} &= t \\ (\!| [t_1, \ldots, t_n] |\!)_{R,i} &= t_i \\ (\!| e_1\ e_2 |\!)_{R,i} &= (\!| e_1 |\!)_{R,i}\ (\!| e_2 |\!)_{R,i} \end{aligned}$$

Finally, assuming a top-level program p is closed, then given a role R and node i for that role, we can form a closed Async node program for the initial state of node i by computing $(\!| p |\!)_{R,i}$.

As an example, the Async node programs for the two roles in the `SimpleVote` program in Figure 1a are shown below, using the notation $\mathsf{let}\ x \coloneqq e_1\ \mathsf{in}\ e_2$ to represent $\mathsf{bind}\ e_1\ (\lambda\, x.e_2)$:

Async node program for the leader:	*Async node program for the (correct) replicas:*
$\lambda p.$ let `cnt` $\coloneqq$ receive c 0 (`fcnteq` p) in return (`calc_dec cnt` x)	$\lambda x.$ send $c\ x$

The execution of a single Async node program requires an environment that includes other nodes and some model that captures the intuitive behavior for communication mentioned above. To achieve this, our next step is to build a transition system that models communication channels and then to compose Async node and channel transition systems into a global transition system.

4.3 Single-use Channel Semantics

A *channel* is an abstraction used to describe the network. Instead of being modeled as a single entity, the network is logically divided into separate communication channels, each serving only a fixed set of senders and receivers. Channels simplify reasoning by separating messages sent for different logical purposes. We model network behaviors using multiple single-use channels instead of the more common multi-use channels. Each single-use channel models a single logical round of communication, and different channels correspond to different logical communication steps. We associate channels with statically known information and give them a dynamic operational semantics. The list of parameters that defines a channel is as follows:

Definition 4.4 (Async Channel Parameters)**.** Assume we are given a role set $\mathcal{R}$, a configuration C for $\mathcal{R}$, and a channel context Δ. Recall that each channel appears at most once in Δ and is associated with a message type τ, as well as a sender role and a receiver role, which can be the same role. We use $\mathsf{msg\_t}_\Delta(c)$ to represent the type τ associated with c in Δ.

Suppose c is a channel associated with sender role S and receiver role R. Then $\mathsf{sender}(c)$ denotes the set of non-Byzantine nodes in configuration C associated with role S; $\mathsf{Byz\_sender}(c)$ denotes

the set of Byzantine sender nodes in C associated with role S; and receiver(c) denotes the set of non-Byzantine receiver nodes in C associated with role R.

We also define $\mathsf{Netwk}_{\mathsf{Async}}$, which models the effect of asynchrony of the network in the same language of Netwk_R.

$$\mathsf{Netwk}_{\mathsf{Async}} : \mathsf{list\ msg\_t}(c) \to \mathcal{P}(\mathsf{list\ msg\_t}(c)) \coloneqq \lambda l.\ (\mathsf{perm}\ l) \gg\!= (\lambda l.\ \mathsf{trunc}\ l\ (n_S - f_S))$$

$\mathsf{Netwk}_{\mathsf{Async}}$ is a function that maps a list of messages sent to a receiver node to the set of possible lists of messages the receiver could possibly receive. Under our network assumptions, it only performs two operations: permutation, which models message reordering in transit, and truncation of the list to a prefix of at least the lower bound on the number of correct sender nodes, i.e., the nodes that are neither Byzantine nor crash. The effects of Byzantine nodes are separately captured by the transition relation.

Given a channel c with the parameters defined above, we define the semantics of Async channel as the labeled transition system $\langle S_c, \Lambda_c, \to_c \rangle$ below.

Definition 4.5 (Async Channel Semantics)**.** A channel state $s \in S_c$ is a tuple of four components:

(1) $F_s \subseteq \mathsf{sender}(c)$, the set of non-Byzantine senders who have performed their send action.
(2) $F_r \subseteq \mathsf{receiver}(c)$, the set of non-Byzantine receivers who have done their receive action.
(3) $F_b : \mathsf{receiver}(c) \to \mathcal{P}(\mathsf{Byz\_sender}(c))$, for each non-Byzantine receiver node, a set of Byzantine sender nodes who have sent the receiver a message.
(4) $M : \mathsf{receiver}(c) \to \mathsf{list\ msg\_t}(c)$, for each receiver a list of message payloads that have been sent to it.

Λ_c has three kinds of labels:

$$l_c ::= \mathsf{send}\ s\ v \mid \mathsf{byz\_send}\ sb\ r\ v \mid \mathsf{receive}\ r\ \vec{v}$$

where $s \in \mathsf{sender}(c)$, $sb \in \mathsf{Byz\_sender}(c)$, $r \in \mathsf{receiver}(c)$, $v : \mathsf{msg\_t}(c)$, and $\vec{v} : \mathsf{list\ msg\_t}(c)$.

send $s\ v$ models a non-Byzantine sender node s broadcasting a message v to all the receiver nodes. receive $r\ \vec{v}$ models a non-Byzantine receiver node r receiving a well-formed list of messages $\vec{v}$. byz_send $sb\ r\ v$ models a Byzantine sender node sb sends a single message of content v to a single non-Byzantine receiver node r.

$\to_c$ has one rule for each kind of label.

$$\textsc{send}\ \frac{s \notin F_s \wedge F_s' = F_s \cup \{s\} \wedge \forall r, M'(r) = M(r) +\!\!+ [v]}{\langle F_s, F_r, F_b, M\rangle \xrightarrow{\mathsf{send}\ s\ v} \langle F_s', F_r, F_b, M'\rangle}$$

$$\textsc{byz\_send}\ \frac{\begin{array}{c}\forall r' \neq r, F_b'(r') = F_b(r') \wedge M'(r') = M(r') \\ sb \notin F_b(r) \wedge F_b'(r) = F_b(r) \cup \{sb\} \wedge M'(r) = M(r) +\!\!+ [v]\end{array}}{\langle F_s, F_r, F_b, M\rangle \xrightarrow{\mathsf{byz\_send}\ sb\ r\ v} \langle F_s, F_r, F_b', M'\rangle}$$

$$\textsc{receive}\ \frac{r \notin F_r \wedge (r \notin \mathsf{sender}(c) \vee r \in F_s) \wedge F_r' = F_r \cup \{r\} \wedge \vec{v} \in \mathsf{Netwk}_{\mathsf{Async}}\ M(r)}{\langle F_s, F_r, F_b, M\rangle \xrightarrow{\mathsf{receive}\ r\ \vec{v}} \langle F_s, F_r', F_b, M\rangle}$$

Async channel's state and transitions are mostly about bookkeeping of which nodes have performed their send and receive actions to prevent duplicates: each non-Byzantine sender node may only broadcast once; each receiver node may only receive once; and each Byzantine sender

node can only send to each receiver node at most once. In addition, if the sender role is the same as the receiver role, which means a sender node is also a receiver node, it must send before it receives.

In the receive transition rule, we account for network-reordering and dropping effects with $\mathsf{Netwk}_{\mathsf{Async}}$. The initial state of the channel is the tuple $\langle \emptyset, \emptyset, \lambda\ \_.\emptyset, \lambda\ \_.[] \rangle$.

4.4 Composing Labeled Transition Systems

We build a *global* transition system from a set of local Async node and channel transition systems using the following notions of interaction composition:

Definition 4.6 (Interaction Composition)**.** Let $\mathcal{T}_1 = \langle S_1, \Lambda_1, \rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, \Lambda_2, \rightarrow_2 \rangle$ be labeled transition systems, and let $\Lambda \subseteq (\Lambda_1 + \mathbb{1}) \times (\Lambda_2 + \mathbb{1})$. Then the interaction composition $\mathcal{T}_1 \bowtie_\Lambda \mathcal{T}_2$ is defined to be $\langle S_1 \times S_2, \Lambda, \rightarrow \rangle$ where $(s_1, s_2) \xrightarrow{(\ell_1, \ell_2)} (s_1', s_2')$ when (a) $\ell_1 = \mathtt{tt}$ and $s_1' = s_1$ or $s_1 \xrightarrow{\ell_1}_1 s_1'$, and (b) $\ell_2 = \mathtt{tt}$ and $s_2' = s_2$ or $s_2 \xrightarrow{\ell_2}_2 s_2'$.

Note that we must specify a subset of permissible labels via Λ. In other words, the choice of Λ delimits the permissible combinations of local labels and thus can be used to model the interactions between the local systems. For instance, consider a system composed of a sender and a receiver. The only label for the sender is send x, for any integer x. The only label for the receiver is receive y, for any integer y. By only allowing the global labels to be of the form (send z, receive z) for the same integer z, we can model synchronous communication between the sender and the receiver.

Given the definition of interaction composition, a special case is when the sub-systems do not interact:

Definition 4.7 (Non-Interacting Composition)**.** Let $\mathcal{T}_1 = \langle S_1, \Lambda_1, \rightarrow_1 \rangle$ and $\mathcal{T}_2 = \langle S_2, \Lambda_2, \rightarrow_2 \rangle$ be labeled transition systems. Then the non-interacting composition $\mathcal{T}_1 \otimes \mathcal{T}_2$ is given by $\mathcal{T}_1 \bowtie_\Lambda \mathcal{T}_2$, where the labels in Λ are either of the form $(\ell_1, \mathtt{tt})$ or $(\mathtt{tt}, \ell_2)$.

In other words, the non-interacting composition only allows local transitions. When we have an indexed sequence of transition systems $[\mathcal{T}_1, \ldots, \mathcal{T}_n]$, we write $\otimes_i \mathcal{T}_i$ to represent $\mathcal{T}_1 \otimes \cdots \otimes \mathcal{T}_n$. If $s = (s_1, \ldots, s_n)$ is a state in a composed system, we write $s@i$ to represent s_i. Similarly, if ℓ is a label in the product, then $\ell@i$ represents the label's i^{th} component.

Definition 4.8 (Global Compilation)**.** Given a well-typed program $\Delta; \emptyset \vdash_{\mathcal{R}} p : \tau$, and a configuration C, we define the *global* compilation of p to be:

$$(\!|\, p \,|\!) = (\otimes_{R, i \in R} (\!|\, p \,|\!)_{R,i}) \bowtie_\Lambda (\otimes_{c \in \mathrm{Dom}(\Delta)} (\!|\, c \,|\!))$$

where $(\!|\, c \,|\!)$ for a channel c is the initial channel state of the transition system described in Section 4.3, and where we write $i \in R$ to mean i is the node identifier in the set of good nodes for role R specified by configuration C. Additionally, we must specify Λ, the valid global labels for the composed system. A global label ℓ must be of one of the three forms:

- Send: for some c, p, and v, $\ell@c$ = send p v, $\ell@p$ = send c v, and for all j, $j \neq p \wedge j \neq c$, $\ell@j = \mathtt{tt}$.
- Byzantine: for some c, p, q, and v, $\ell@c$ = byz_send p q v, and for all $j \neq c$, $\ell@j = \mathtt{tt}$.
- Receive: for some c, p, and $\vec{v}$, $\ell@p$ = receive c $\vec{v}$ and $\ell@c$ = receive p $\vec{v}$, and for all j, $j \neq p \wedge j \neq c$, $\ell@j = \mathtt{tt}$.

This choice of global labels assumes that no more than one node and exactly one channel can take a local step in each discrete global step. The send label and receive label model the interaction between a single non-Byzantine node and a single channel. The two local labels must agree on the payload of the message(s) being sent/received. The byz_send label models a stand-alone Byzantine

action where an arbitrary message is sent from a Byzantine node to a non-Byzantine receiver through a channel. The asynchronous communication between two nodes is modeled in this way as two separate synchronous steps between a node and a channel.

The state and the transition relation of this compiled system follow from the construction. We also have a unique initial state s_0 for the global system: Async node components map to their initial Async node program, while Async channel components map to the empty channel state.

We say an Async system state s_f is *completed* if all its Async node instances have all been reduced to a return.[3] We write $s{\downarrow}$ as the partial function which extracts these values and returns them as a record of vectors $\vec{v}$, such that $\vec{v}@i = v$ iff $s@i = \mathsf{return}\ v$ for any node i. For an Async system instance compiled from a Sync program, given any completed state s_f, we can turn $s_f{\downarrow}$ into a record value of type $[\![\tau]\!]$, where the different values are broken out by role. We define $(\!|\, p \,|\!){\downarrow}$ to be the set of all such output values of the completed states reachable from the initial state. This is the set of possible outputs of the program p in the Async (system) semantics.

4.4.1 Properties. A critical aspect of this construction is the relation between global behaviors (traces) and local views of behaviors. In particular, we make use of two key lemmas: a decomposition lemma that shows all global behaviors are necessarily arrangements of local behaviors, and a composition lemma that shows we can always assemble traces from local systems into a global trace when there are suitable global labels.

We first define the projection from a sequence of global labels to sequences of local labels.

Definition 4.9 (Global Label Sequence Projections)**.** Given a sequence of global labels $L \in (\Lambda_G)^*$ and a local system identifier i, the subsequence of global labels related to i is defined as follows:

$$\mathsf{filter}_i(L) \coloneqq \begin{cases} [] & L = [] \\ \mathsf{filter}_i(L') & L = l :: L' \wedge l@i = \mathtt{tt} \\ l :: \mathsf{filter}_i(L') & L = l :: L' \wedge l@i \neq \mathtt{tt} \end{cases}$$

The projection of L to a sequence of i-local labels is defined as follows:

$$\mathsf{proj}_i(L) \coloneqq \mathsf{map}\ (@i)\ \mathsf{filter}_i(L)$$

Assume a system $\mathcal{T} = \mathcal{T}_1 \bowtie_\Lambda \mathcal{T}_2$, where $\mathcal{T}_i = \langle S_i, \Lambda_i, \rightarrow_i \rangle$.

Lemma 4.1 (Decomposition)**.** Given a sequence of labels $L \in \Lambda^*$ and compound states s and t such that $s \xrightarrow{L} t$, then $s@i \xrightarrow{\mathsf{proj}_i(L)}_i t@i$.

Lemma 4.2 (Composition)**.** Given component label sequences $L_i \in \Lambda_i^*$ and states s_i, t_i such that $s_i \xrightarrow{L_i}_i t_i$, and a compound label sequence L such that $\mathsf{proj}_i(L) = L_i$, then $(s_1, s_2) \xrightarrow{L} (t_1, t_2)$.

Lemma 4.1 implies that any property proven for all possible behaviors of local systems is preserved by the global system. Intuitively, a local system is most "free" when the environment it is interacting with is arbitrary. Composing it with other systems provides more information about the environment and restricts its possible behaviors.

Lemma 4.2 shows that any combination of possible local system behaviors can happen in the global system if there exist suitable global labels. This can be used to prove a sequence of global labels are *permissible* (*i.e.,* lead from one global state to another) by establishing all its projections are permissible local label sequences.

[3]Note that this may not be a terminal state in the transition system, as Byzantine sends could happen on channels.

5 Adequacy Proof

In this section, we sketch the proof that our denotational semantics is adequate for the compiled Async system operational semantics.

For any closed Sync program $\Delta; \emptyset \vdash_{\mathcal{R}} p : \tau$, $[\![p]\!]$ produces a set of possible outputs of type $[\![\tau]\!]$. The program also compiles to an Async system and yields a set of possible outputs of that system $(\!|\, p \,|\!)\downarrow$.

Theorem 5.1 (Adequacy). $(\!|\, p \,|\!)\downarrow \subseteq [\![p]\!]$.

We prove the above result in two steps:

Step 1: By definition, any possible output of an Async system is witnessed by a trace L such that there exists a completed state s_f which gives the output and $s_0 \xrightarrow{L} s_f$. We show that we can reduce the set of traces to be considered in the operational semantics to an "aligned" subset that contains the same outputs and whose execution order puts all of the operations on a given channel together.

Step 2: We prove $s_f\downarrow \in [\![p]\!]$ assuming L is an aligned trace.

5.1 Reduction to Aligned Traces

We begin by defining the notion of the *alignment* of a global trace with respect to a channel context. Intuitively, this captures the idea that the outputs of a given trace are the same as a trace that forces all of the interactions to happen in an order specified by Δ.

Definition 5.1 (Alignment of a Global Trace). Given a channel context Δ and a sequence of global Async system labels L, the alignment of L with respect to Δ is given by:

$$\mathsf{align}_\Delta(L) \coloneqq \mathsf{flatmap}\ (\lambda c.\ \mathsf{filter}_c(L))\ \mathsf{Dom}(\Delta)$$

That is, $\mathsf{align}_\Delta(L)$ takes a list of labels, and for each channel c, projects out the sub-list of labels corresponding to non-empty transitions for c, and then concatenates those lists back together in the order that the channels are listed in Δ. Note that for every global label, there is a unique channel that transitions. Thus, $\mathsf{align}_\Delta(L)$ produces a permutation of the labels in L.

For example, consider a system with two good nodes n_1 and n_2 with n_1 sending messages to n_2 through channel c_1 followed by c_2, and a Byzantine node b sending a message on c_1. One possible sequence of actions we could see is the following

$$[n_1(\mathsf{send}\ c_1\ v_1); n_1(\mathsf{send}\ c_2\ v_2); n_2(\mathsf{receive}\ c_1\ \vec{v_1}); n_2(\mathsf{receive}\ c_2\ \vec{v_2}); b(\mathsf{byz\_send}\ c_1\ w)]$$

where we only show the relevant node-portions of the labels. After aligning the trace, we will have:

$$[n_1(\mathsf{send}\ c_1\ v_1); n_2(\mathsf{receive}\ c_1\ \vec{v_1}); b(\mathsf{byz\_send}\ c_1\ w); n_1(\mathsf{send}\ c_2\ v_2); n_2(\mathsf{receive}\ c_2\ \vec{v_2})]$$

So that all of the c_1 actions are performed before the c_2 actions, but otherwise the relative order of transitions is preserved.

Theorem 5.2 (Aligned Trace Permissibility). Let s_0 be the initial state of an Async system built from a well-typed Sync program p with the channel context Δ and suppose $s_0 \xrightarrow{L} s$. Then $s_0 \xrightarrow{\mathsf{align}_\Delta(L)} s$.

Proof. We use the composition lemma (Lemma 4.2) to prove $\mathsf{align}_\Delta(L)$ is a permissible trace of the system. This requires us to prove that any local projection of $\mathsf{align}_\Delta(L)$ to a local system is a permissible trace of that local system. By the decomposition lemma (Lemma 4.1), we know any projections of the given trace L are permissible. So it suffices to prove that any local projection of $\mathsf{align}_\Delta(L)$ is equivalent to that of L.

In an Async system, we have two kinds of local systems: channels and nodes. Recall that $\mathsf{align}_\Delta(L)$ reorders the labels in the channel order captured by Δ, so it preserves the projections to the channels

naturally. More formally, for any channel c in the system, we have:

$$\begin{aligned}
\mathsf{proj}_c(\mathsf{align}_\Delta(L)) &= \mathsf{map}\ (@c)\ (\mathsf{filter}_c(\mathsf{flatmap}\ (\lambda c'.\mathsf{filter}_{c'}(L))\ \mathsf{Dom}(\Delta))) && \backslash\backslash \text{ unfold definitions}\\
&= \mathsf{map}\ (@c)\ (\mathsf{flatmap}\ (\lambda c'.\mathsf{filter}_c(\mathsf{filter}_{c'}(L)))\ \mathsf{Dom}(\Delta)) && \backslash\backslash \text{ list properties}\\
&= \mathsf{map}\ (@c)\ (\mathsf{filter}_c(L)) && \backslash\backslash \text{ other channels give []}\\
&= \mathsf{proj}_c(L) && \backslash\backslash \text{ by definition}
\end{aligned}$$

For nodes, we begin by defining a relation ($\sim_R$) that holds when a sequence of transition labels respects the order in Δ according to a given role R:

$$\begin{aligned}
[] &\sim_R \Delta\\
(\mathsf{send}\ c\ v) :: (\mathsf{receive}\ c\ \vec{v}) :: L &\sim_R (c : (R, R, \tau)) :: \Delta \text{ when } L \sim_R \Delta\\
(\mathsf{send}\ c\ v) :: L &\sim_R (c : (R, S, \tau)) :: \Delta \text{ when } L \sim_R \Delta\\
(\mathsf{receive}\ c\ \vec{v}) :: L &\sim_R (c : (S, R, \tau)) :: \Delta \text{ when } L \sim_R \Delta \wedge R \neq S\\
L &\sim_R (c' : (S_1, S_2, \tau)) :: \Delta \text{ when } L \sim_R \Delta \wedge R \neq S_1 \wedge R \neq S_2
\end{aligned}$$

We then extend this relation to describe when an Async node program respects Δ. Our intuition is that all of the possible traces the program can generate should respect the ordering in Δ:

$$\begin{aligned}
\mathsf{return}\ v &\sim_R \Delta \text{ when } R \notin \Delta\\
\mathsf{sendThen}\ c\ v\ (\mathsf{rcvThen}\ c\ k) &\sim_R (c : (R, R, \tau)) :: \Delta \text{ when } \forall \vec{v}, k(\vec{v}) \sim_R \Delta\\
\mathsf{sendThen}\ c\ v\ k &\sim_R (c : (R, S, \tau)) :: \Delta \text{ when } k \sim_R \Delta \wedge R \neq S\\
\mathsf{rcvThen}\ c\ k &\sim_R (c : (S, R, \tau)) :: \Delta \text{ when } \forall \vec{v}, k(\vec{v}) \sim_R \Delta\\
t &\sim_R (c' : (S_1, S_2, \tau)) :: \Delta \text{ when } t \sim_R \Delta \wedge R \neq S_1 \wedge R \neq S_2
\end{aligned}$$

Lemma 5.3 (Compilation Respects Channel Ordering)**.** Suppose $\Delta; \emptyset \vdash_{\mathcal{R}} p : \tau$, and let $R \in \mathcal{R}$, $i \in R$. Let L be a sequence of Async node labels and t an Async node program such that $(\!|\, P \,|\!)_{R,i} \xrightarrow{L} t$. Then $L \sim_R \Delta$.

Proof. It is easy to see by induction on the typing derivation for p that $(\!|\, p \,|\!)_{R,i} \sim_R \Delta$. We argue that for any Async node program t such that $t \sim_R \Delta$, that if $t \xrightarrow{L} t'$, then $L \sim_R \Delta$, and furthermore, there exists a suffix of Δ, Δ' such that $t' \sim_R \Delta'$. The argument proceeds by induction on the length of L and then via case analysis on the structure of t. □

Now for any role R and non-Byzantine node i in that role, let $L_i = \mathsf{proj}_i(L)$. By Lemma 5.3, we know that $L_i \sim_R \Delta$ and we need to show $L_i = \mathsf{proj}_i(\mathsf{align}_\Delta(L))$. The proof proceeds by induction on the derivation of $L_i \sim_R \Delta$. □

5.2 Adequacy of Aligned Traces

We now prove Theorem 5.1 by showing that the output of any completed and aligned trace belongs to the set of outputs given by the denotational semantics.

Formally, we show for any closed Sync program P, well-typed with $\Delta; \Gamma \vdash_{\mathcal{R}} P : \tau$,

$$(\!|\, P \,|\!)\downarrow \subseteq [\![P]\!]$$

By definition of $\downarrow$, for any output $o \in (\!|\, P \,|\!)\downarrow$, there exists a trace l and a state F such that F is a complete state, $F\downarrow = o$, and $(\!|\, P \,|\!) \xrightarrow{l} F$.

By the alignment theorem, we can assume $l = \mathsf{align}_\Delta(l)$ without loss of generality.

Thus, assume $\Delta = [c_1, c_2, \ldots, c_n]$, $l = l_1 +\!\!+\ l_2 +\!\!+\ \ldots l_n$, where l_i contains only symbols associated with channel c_i.

Additionally, we can assume P is in the let-normal form:

$$\begin{aligned} P = \ & \texttt{let } x_1 \coloneqq \texttt{comm } c_1\ m_1\ d_1\ f_1 \texttt{ in} \\ & \texttt{let } x_2 \coloneqq \texttt{comm } c_2\ m_2\ d_2\ f_2 \texttt{ in} \\ & \ldots \\ & \texttt{let } x_n \coloneqq \texttt{comm } c_n\ m_n\ d_n\ f_n \texttt{ in} \\ & \texttt{ret } \{R_i \mapsto e_i\} \end{aligned}$$

We now define a big-step operational semantics for any closed, well-typed, and normalized Sync program P with configuration C.

$$\frac{\Delta(c) = (S, R, \tau_m) \quad L = [l_1, l_2, \ldots, l_{n_R - b_R}] \quad \forall i, l_i \in \mathsf{Netwk}_S([\![m]\!]) \quad y = \{R \mapsto \mathsf{map3}\ \mathsf{foldl}\ [\![f]\!]\ [\![d]\!]\ L\}}{\texttt{let } x \coloneqq \texttt{comm } c\ m\ d\ f \texttt{ in } P' \xrightarrow{L} P'[y/x]}$$

We use $\langle P \rangle\!\downarrow$ to denote the set of possible outputs in this big-step semantics, defined as follows:

$$\langle \texttt{ret } \{R_i \mapsto e_i\} \rangle\!\downarrow = \mathsf{singleton}\{R_i \mapsto [\![e_i]\!]\}$$

$$\langle \texttt{let } x \coloneqq \texttt{comm } c\ m\ d\ f \texttt{ in } P'_x \rangle\!\downarrow = \bigcup_{P \xrightarrow{L} P'} \langle P' \rangle\!\downarrow$$

This splits the proof of the adequacy theorem into two parts $(\!|\, P \,|\!)\!\downarrow \subseteq \langle P \rangle\!\downarrow$ and $\langle P \rangle\!\downarrow \subseteq [\![P]\!]$.

5.2.1 Big-step to Denotational. For $\langle P \rangle\!\downarrow \subseteq [\![P]\!]$, we perform an induction on the structure of P.

Proof. Base case: when $P = \mathsf{ret}\{R_i \mapsto e_i\}$, $\langle P \rangle\!\downarrow = \mathsf{singleton}\{R_i \mapsto [\![e_i]\!]\} = [\![P]\!]$.

Inductive case: when $P = \texttt{let } x \coloneqq \texttt{comm } c\ m\ d\ f \texttt{ in } P'_x$. By definition of the big-step semantics, $\langle P \rangle\!\downarrow = \bigcup_{P \xrightarrow{L} P'} \langle P' \rangle\!\downarrow$. It suffices to prove for any L and P' such that $P \xrightarrow{L} P'$, $\langle P' \rangle\!\downarrow \subseteq [\![P]\!]$.

By the induction hypothesis, we know $\langle P' \rangle\!\downarrow \subseteq [\![P']\!]$. It is enough to prove $[\![P']\!] \subseteq [\![P]\!]$.

This follows from the definition of the big-step semantics and the denotational semantics. □

5.2.2 Async to Big-step. To move from Async traces to Big-step traces, we first need a well-formedness property of Async traces.

Definition 5.2 (Finished channel)**.** Recall that a state of an Async channel c is a tuple $\langle F_s, F_r, F_b, M \rangle$, where F_s is the set of non-Byzantine senders and F_r is the set of non-Byzantine receivers.

A state is *finished*, if and only if $F_s = \mathsf{sender}(c)$ and $F_r = \mathsf{receiver}(c)$. This means each non-Byzantine sender has performed their send operation, and each non-Byzantine receiver has performed their receive operation.

The first lemma we need here is that all channels are finished in a complete Async trace.

Proof Sketch: In a complete trace, each node must be complete. Because the compilation guarantees each node respects the channel context, Δ, for each channel, each of the non-Byzantine senders and receivers must have performed their send and receive actions on this channel. Thus, every channel is finished.

For convenience, we extend the definition of Async channel state with two extra bookkeeping states. $M_s : \mathsf{list}\ \tau_m$ records each message sent by each non-Byzantine sender node. $M_r : F_r \to \mathsf{list}\ \tau_m$ records the list of messages received by each non-Byzantine receiver node. We modify the transitions accordingly to put information into those two pieces of the states.

Importantly, with the extra information, we can extract a big-step label $L = [l_1, l_2, \ldots, l_{n_R - b_R}]$ from a finished channel state by having $l_i = M_r^f(r_i)$, where M_r^f is the M_r of the finished state and r_i is a specific node of the receiver role.

So the second lemma we need is that this always gives a valid L such that $\forall i, l_i \in \mathsf{Netwk}_S(M_s^f)$. M_s^f is the M_s of the finished state and contains the messages sent by all non-Byzantine sender nodes.

Proof Sketch: We do an induction over the Async channel trace. By definition, each received list of messages satisfies Netwk_{Async} with the list of messages that have been sent to the receiver at the time of the receive action. We need to prove that each received list of messages satisfies $\mathsf{Netwk}_S(M_s^f)$. This can be proven with the invariant that $M(r)$ is always a subset of add_any $M_s^f\ b_R$ and thus $\mathsf{Netwk}_{Async}(M(r)) \subseteq \mathsf{Netwk}_S(M_s^f)$.

A third lemma is needed to tie the changes in the node state to the actions that happened in the channel. More specifically, a node that sends a message will always send the message described by its program and continue with a unit; a node that receives a list of messages will continue with the result of folding its handler over the list of received messages. This lemma can be proven following the definitions of the Async transitions.

The last lemma allows us to erase finished channels from the Async state. For any Async trace $S \xrightarrow{l} F$ and S contains a finished channel state $S@c$, then $S' \xrightarrow{l'} F'$, where S' is S except $S@c$, F' is F except $F@c$, and l' is l without labels associated with c. This lemma holds because the only transitions in l associated with c are Byzantine send operations on channel c due to $S@c$ being in a finished state and these operations can only affect the state of c.

To prove $(\!|\, P \,|\!)\downarrow \subseteq \langle P \rangle\downarrow$, we perform an induction on the structure of P.

Proof. Base case: when $P = \mathsf{ret}\{R_i \mapsto e_i\}$, by the definition of the compilation rules, both sides are the singleton set of $\{R_i \mapsto [\![e_i]\!]\}$.

Inductive case: when $P = \texttt{let}\ x \coloneqq \texttt{comm}\ c\ m\ d\ f\ \texttt{in}\ P'_x$. It suffices to show, for any Δ-aligned Async trace $(\!|\, P \,|\!) \xrightarrow{l} F$, $F\downarrow \in \langle P \rangle\downarrow$.

We consider a block of labels of the same channel at a time. Because l is Δ-aligned, it must start with a block of labels associated with c, so $l = l_c +\!\!+ l'$ and $(\!|\, P \,|\!) \xrightarrow{l_c} S \xrightarrow{l'} F$. By the first lemma above, because F is a completed state, it contains a finished channel c state $F@c$. By the definition of alignment, l' contains no labels associated with c, so $S@c = F@c$ and $S@c$ is a finished channel state. By the second lemma above, we can extract a valid big-step label L from $S@c$. This gives us a new Sync program P' through $P \xrightarrow{L} P'$. By the third lemma above, the node states in S match those of $(\!|\, P' \,|\!)$. They only differ because P' does not have channel c. And by the fourth lemma, there exists F', such that $(\!|\, P' \,|\!) \xrightarrow{l'} F'$. We know $F'\downarrow = F\downarrow$ because the only difference between F and F' is that F includes channel c's state and F' does not, and the extracted output does not depend on the channel states. By the induction hypothesis, $F'\downarrow \in \langle P' \rangle\downarrow$. By the definition of the big-step semantics, $\langle P' \rangle\downarrow \subseteq \langle P \rangle\downarrow$. So we conclude $F\downarrow \in \langle P \rangle\downarrow$. □

6 Related Work

Despite our goal being to explore new directions for formal reasoning of distributed systems rather than pushing the state-of-the-art of what can be formally verified, our work is closely related to formal verification frameworks of distributed systems.

Reducing Asynchrony to Synchrony. The line of work on the Heard-Of model [Charron-Bost et al. 2011; Charron-Bost and Schiper 2009; Damian et al. 2019; Drăgoi et al. 2016] also observes that certain asynchronous protocols can be verified by reasoning about their synchronous counterparts.

[Charron-Bost et al. 2011] formalizes the model in Isabelle/HOL. PSync [Drăgoi et al. 2016] is embedded in Scala and provides a high-level, synchronous language for describing crash fault-tolerant systems, which can be executed on partially asynchronous networks with a runtime. [Damian et al. 2019] proposes and formalizes the concept of *communication-closed* protocols, in which communication happens in rounds and messages are either received in that round or not at all. It then generalizes the asynchrony to synchrony reduction to such protocols.

While we offer a functional semantics for compositional theorem-proving in Rocq, the Heard-Of model is mostly adopted for automated verification of manually annotated imperative programs. Furthermore, as a language-based approach, well-typed Sync programs are adequate for reasoning by construction, while in [Damian et al. 2019], the communication-closed restrictions are enforced by manually annotating C programs and then checked by a stand-alone checker tool.

Modeling and Reasoning about Byzantine Failures. We model both crash failures and Byzantine failures through the unified Netwk abstraction. Velisarios [Rahli et al. 2018] and its successor, Asphalion [Vukotic et al. 2019], are verification frameworks built in Rocq that also support reasoning about Byzantine behaviors. Using a state machine model and the logic of events [Bickford et al. 2012], the protocols need to be reasoned about at a relatively low level similar to that of Async. However, the frameworks provide epistemic models for the Byzantine adversary and built-in primitives for cryptographic signatures. The authors used the framework to verify and produce an executable artifact of the PBFT protocol [Castro and Liskov 1999].

Layered Refinements. An enabling technique for many existing works that use trace-based semantics is *layered refinement*. Layered refinements have been shown to be effective at reducing the complexity of those proofs. LiDODAG [Qiu et al. 2025] and its predecessors [Honoré et al. 2021, 2022; Qiu et al. 2024] are Rocq-based verification frameworks for distributed systems that provide several state machine-based trace semantics at different abstraction levels. For instance, the top-level trace models the runtime behavior operationally as a non-deterministically growing tree, while at lower levels, the system state contains more details, such as program counters for each node and network messages. The authors verified a handful of consensus protocols with more advanced features, such as DAG-based consensus and dynamic reconfiguration, using their layered semantics by modeling it at those different levels and proving refinement relations between them. Our compilation from Sync to Async has a similar flavor to that of layer refinement, with two differences: (1) The relationship between Sync and Async is proven once and for all, instead of for every verification instance; and (2) Sync's semantics is functional and can be decomposed along its syntax, while layered refinement can only move from one state transition system to another.

Separation Logic. Distributed systems that assume more benign execution environments can adopt off-the-shelf techniques such as separation logic for reasoning and provide richer node-local semantics. Aneris [Krogh-Jespersen et al. 2020] is a Rocq framework that utilizes the concurrent separation logic framework Iris [Jung et al. 2015] to provide local reasoning for the nodes involved. It supports higher-order store and network sockets and has been used to verify a load balancer with concurrent local programs. Disel [Sergey et al. 2017] is also a separation logic-based framework in Rocq and supports compositional reasoning about interactions between an abstract core distributed system and multiple clients. Grove [Sharma et al. 2023] is another concurrent separation logic library in Rocq. It supports many system features, including time-based leases, reconfiguration, crash recovery, and thread-level concurrency.

Decidable Logics. Another approach that eases the proof burden is to restrict the implementation and the specification such that the verification problem falls under some decidable logic and can be solved automatically, as shown in Ivy [Taube et al. 2018], TIP [Berkovits et al. 2019], and also adopted in Verus [Lattuada et al. 2024]. The design of these languages explores a direction different from ours, where a high degree of automation can be achieved through SMT or specialized solvers.

References

Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. arXiv:1712.01367 [cs] doi:10.48550/arXiv.1712.01367

Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2019. Sync HotStuff: Simple and Practical Synchronous State Machine Replication.

Bowen Alpern and Fred B. Schneider. 1987. Recognizing Safety and Liveness. *Distributed Computing* 2, 3 (Sept. 1987), 117–126. doi:10.1007/BF01782772

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. doi:10.1007/978-3-030-25543-5_15

Mark Bickford, Vincent Rahli, and Robert L. Constable. 2012. Logic of Events, a Framework to Reason about Distributed Systems.

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] doi:10.48550/arXiv.2003.03052

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On Global Types and Multi-party Sessions. In *Formal Techniques for Distributed Systems*, Roberto Bruni and Juergen Dingel (Eds.). Springer, Berlin, Heidelberg, 1–28. doi:10.1007/978-3-642-21461-5_1

Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*. USENIX Association, New Orleans, LA, 173–186.

K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* 3, 1 (Feb. 1985), 63–75. doi:10.1145/214451.214456

Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. 2011. Formal Verification of Consensus Algorithms Tolerating Malicious Faults. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer, Berlin, Heidelberg, 120–134. doi:10.1007/978-3-642-24550-3_11

Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of Model: Computing in Distributed Systems with Benign Faults. *Distributed Computing* 22, 1 (April 2009), 49–71. doi:10.1007/s00446-009-0084-6

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 143–156. doi:10.1145/1411204.1411226

Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. 2019. Communication-Closed Asynchronous Protocols. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 344–363. doi:10.1007/978-3-030-25543-5_20

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 (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 400–415. doi:10.1145/2837614.2837650

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. doi:10.1145/3149.214121

Eli Goldweber, Nuda Zhang, and Manos Kapritsos. 2020. Brief Announcement: On the Significance of Consecutive Ballots in Paxos. In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC '20)*. Association for Computing Machinery, New York, NY, USA, 172–174. doi:10.1145/3382734.3405700

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 (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428

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. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 97:1–97:31. doi:10.1145/3485474

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 (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 379–394. doi:10.1145/3519939.3523444

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *ACM SIGPLAN Notices* 50, 1 (Jan. 2015), 637–650. doi:10.1145/2775051.2676980

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems* 27, 4 (Jan. 2010), 7:1–7:39. doi:10.1145/1658357.1658358

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*

(Lecture Notes in Computer Science), Peter Müller (Ed.). Springer International Publishing, Cham, 336–365. doi:10.1007/978-3-030-44914-8_13

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169. doi:10.1145/279227.279229

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA.

Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 370–384. doi:10.1145/3341301.3359651

Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. 2017. *Recovering Shared Objects without Stable Storage (Extended Version)*. Technical Report UW-CSE-TR-17-10-01. University of Washington.

Atsuki Momose and Jason Paul Cruz. 2019. Force-Locking Attack on Sync Hotstuff.

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph. D. Dissertation. IT University of Copenhagen.

Joachim Neu, Ertem Nusret Tas, and David Tse. 2021. Ebb-and-Flow Protocols: A Resolution of the Availability-Finality Dilemma. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, USA, 446–465. doi:10.1109/SP40001.2021.00045

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, USA, 305–320.

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. *Artifact for PLDI 2024 paper #290: LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs*. 8, PLDI (June 2024), 193:1140–193:1164. doi:10.1145/3656423

Longfei Qiu, Jingqi Xiao, Ji-Yong Shin, and Zhong Shao. 2025. LiDO-DAG: A Framework for Verifying Safety and Liveness of DAG-Based Consensus Protocols. *Artifact for PLDI 2025 Paper #316 LiDO-DAG: A Framework for Verifying Safety and Liveness of DAG-based Consensus Protocols* 9, PLDI (June 2025), 203:1392–203:1416. doi:10.1145/3729306

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

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 28:1–28:30. doi: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 (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 113–129. doi:10.1145/3600006.3613172

Yee Jiun Song and Robbert van Renesse. 2008. Bosco: One-Step Byzantine Asynchronous Consensus. In *Distributed Computing (Lecture Notes in Computer Science)*, Gadi Taubenfeld (Ed.). Springer, Berlin, Heidelberg, 438–450. doi:10.1007/978-3-540-87779-0_30

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. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 662–677. doi:10.1145/3192366.3192414

Ivana Vukotic, Vincent Rahli, and Paulo Esteves-Veríssimo. 2019. Asphalion: Trustworthy Shielding against Byzantine Faults. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 138:1–138:32. doi:10.1145/3360564

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. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 357–368. doi:10.1145/2737924.2737958

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Transactions on Computer Systems* 20, 3 (Aug. 2002), 283–328. doi:10.1145/566340.566343