# Programming Language Abstractions for Modularly Verified Distributed Systems


James R. Wilcox[1], Ilya Sergey[2], and Zachary Tatlock[3]

1 University of Washington, Seattle, WA, USA
jrw12@cs.washington.edu
2 University College London, London, UK
i.sergey@ucl.ac.uk
3 University of Washington, Seattle, WA, USA
ztatlock@cs.washington.edu



## Abstract

Distributed systems are rarely developed as monolithic programs. Instead, like any software, these systems may consist of multiple program components, which are then compiled separately and linked together. Modern systems also incorporate various services interacting with each other and with client applications. However, state-of-the-art verification tools focus predominantly on verifying standalone, *closed-world* protocols or systems, thus failing to account for the compositional nature of distributed systems. For example, standalone verification has the drawback that when protocols and their optimized implementations evolve, one must re-verify the entire system from scratch, instead of leveraging compositionality to contain the reverification effort.

In this paper, we focus on the challenge of modular verification of distributed systems with respect to *high-level protocol invariants* as well as for *low-level implementation safety properties*. We argue that the missing link between the two is a programming paradigm that would allow one to reason about both high-level distributed protocols and low-level implementation primitives in a single verification-friendly framework. Such a link would make it possible to reap the benefits from both the vast body of research in distributed computing, focused on modular protocol decomposition and consistency properties, as well as from the recent advances in program verification, enabling construction of provably correct systems implementations. To showcase the modular verification challenges, we present some typical scenarios of decomposition between a distributed protocol and its implementations. We then describe our ongoing research agenda, in which we are attempting to address the outlined problems by providing a typing discipline and a set of domain-specific primitives for specifying, implementing and verifying distributed systems. Our approach, mechanized within a proof assistant, provides the means of decomposition necessary for modular proofs about distributed protocols and systems.



**1998 ACM Subject Classification** D.3.3 [Programming Languages] Language Constructs and Features, F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Programs

**Keywords and phrases** Distributed systems, program verification, distributed protocols, domain-specific languages, type systems, dependent types, program logics

**Digital Object Identifier** 10.4230/LIPIcs.SNAPL.2017.19


## 1 Introduction

As with any software, distributed systems are not built as standalone pieces of code: rather they are assembled from multiple independently developed components. For instance, different nodes may communicate using message passing, components of a particular implementation



2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 19; pp. 19:1–19:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

may be compiled separately, and different systems may interact with each other and with the client applications via regular program flow and by imposing implicit invariants on each other's behavior.

There is a vast amount of work dedicated to establishing and verifying invariants of standalone *distributed protocols*, such as Paxos [23, 24, 61], Raft [46], *etc*, formulated as abstract high-level state-transition systems (see, *e.g.*, [62, 61, 20, 44] for references). Furthermore, several impressive advances have been recently made in verifying specific realistic *systems implementations* with respect to fixed properties [64, 16, 27, 11, 48, 65, 22, 32]. However, the modular nature of these systems is not fully matched by state-of-the art verification techniques, which still follow a "whole-program" approach. Specifically, most of the verification methodologies to date require a complete revision of the proofs (or are not applicable at all) in the following scenarios, which occur regularly in the life cycle of distributed software:

1. A high-level protocol $\mathcal{P}$ (*e.g.*, Paxos) remains the same, but its implementation run by a particular node is updated (*e.g.*, replaced by an optimized one [4]). Naturally, one should now establish that the new implementation *refines* (*i.e.*, exhibits the same externally observable behavior as) the same abstract protocol [1], while all proofs concerning the protocol itself should not change.
2. As a variant of the previous scenario, an optimization in $\mathcal{P}$'s implementation might *delegate* some of the computation to another node, possibly following another protocol $\mathcal{P}'$ [61]. In this case, one should establish that, under certain assumptions about $\mathcal{P}'$, the resulting implementation of $\mathcal{P}$ still refines its specification.
3. An implementation, interacting with other nodes under a protocol $\mathcal{P}$, may make specific assumptions about the initial state of the system, thus restricting the set of reachable states. This is captured by strengthening the protocol's state-space invariant, thus permitting the implementation to leverage additional facts about its state. Such strengthening should *not* cause the proofs of implementations run by other involved nodes to be revised.

The first scenario is fairly standard: one should always be able to make low-level optimizations in an actual implementation, as long as these changes *are not observable* on the abstract level, with the high-level protocol serving as a system specification. The existing solutions [16, 64] for this modularity challenge rely on the classical technique of establishing a *refinement* [1, 26] between an actual implementation (the code) and a specification (a protocol) via forward-backwards simulation [37]. That said, in the presence of program-level composition (*e.g.*, third-party libraries), recursion, and higher-order programming primitives, proving refinement in a modular way becomes a notoriously difficult problem, requiring a non-trivial relational semantics and dedicated program logics. While such logics exist for shared-memory concurrency [59, 30, 56], none exist for distributed systems. The situation is even more complicated in the presence of *fine-grained* communication primitives, such as `send` and `receive` (as opposed to synchronous models [13]), that are used for implementing non-blocking message-passing. To the best of our knowledge there is no program logic that supports reasoning about fine-grained message-passing distributed systems in a modular way, and the state-of-the-art approaches either avoid fine-grained operations all together [10, 11], thus sacrificing potential performance gains, or employ first-order reduction techniques [16, 31, 12].

The second scenario demonstrates an interplay between properties of a protocol and proofs of an implementation that relies on them: indeed, the correctness of a refinement by the latter depends on the invariants of the former. Yet, from a programmer's perspective this is just another program optimization, so the proofs should not be that different from those in the Scenario **1**. However, we are not aware of any verification frameworks allowing one to modularly prove refinement between an implementation and its protocol in this case.

The third scenario demonstrates a common pattern where a protocol implementor assumes the system is initialized to a certain "good" state. This implies any subsequent state of the system is *reachable* from the good state, which can be used to establish additional safety properties. This scenario allows different client implementations using a protocol to rely on different assumptions about its initial conditions and different system invariants. Combined with the second scenario, this means that one should be able to impose custom (but valid) invariants when proving an implementation-specific refinement!

To make things more concrete, let us imagine implementing an optimization of a straightforward distributed computation (*e.g.*, MapReduce), run by a node, that memoizes its past results using some third-party distributed storage. Then, an important invariant of a storage protocol, required for justifying such an optimization, should state that the stored values are never dropped or replaced. However, another client application, which only *queries* the storage but does not *write* into it might be verified under a weaker invariant. From this observation we conclude that *one and the same* distributed protocol might be a subject of different application-specific invariants (since the strongest possible invariant is not alway possible to foresee in advance) and initial state assumptions, but imposing a different inductive invariant should not affect already verified protocol implementations and their proofs.

From the discussion above, it seems that the proofs of refinement, *i.e.*, that an implementation "does not go wrong", are unavoidable for formally establishing the correspondence between the code of an implementation and its abstract protocol specification. In this line of research, in an attempt to overcome the complexity of the refinement proofs, which become especially acute in the presence of horizontal composition of interacting distributed services (*i.e.*, Scenario **2**) and client-specific invariants (*i.e.*, Scenario **3**), we have decided to adopt a different approach for proving programs well-behaved: by means of type theory.

## 2 A Type-Based Approach to Distributed System Verification

We have drawn inspiration from results on Hoare Type Theory (HTT) [43, 42, 41] and specifically its recent variants, which support specifying and verifying fine-grained shared-memory concurrent algorithms [40, 50, 51, 52]. In HTT, an effectful, imperative, potentially higher-order program $e$ is given a Hoare type $\mathsf{HT}\ \{\lambda s.\ A\}\{\lambda r\ s'.\ B\}$, where $A$ is a predicate constraining the pre-state $s$ (*e.g.*, a heap), and $B$ constrains the result $r$ and the post-state $s'$. That is, the pre-/postconditions $A$ and $B$ declaratively specify the effect of $e$ with respect to the state it might affect. Furthermore, the original HTT incorporated Separation Logic-style specifications [41] and adopted *fault-avoiding* semantics [49], thus ensuring that well-typed programs are *memory-safe*. The concurrent extensions of HTT extended the notion of type safety to account for *data race freedom* [28] and *coherence* of a concurrently used resource [40].

**Distributed Hoare Types.** In this work, we extend the notion of Hoare types to distributed system implementations, whose "state" captures both local components (*e.g.*, a heap) and a global component, namely the (multi-)set of messages exchanged by the nodes involved in the system. In this way, "effects" correspond to interactions in a distributed environment between nodes via message passing. Each such interaction (*i.e.*, sending or receiving a message) is synchornized with a change in a node's local state (*e.g.*, updating a set of local permissions). These changes follow one of several available "atomic" transitions, which are provided by user-defined high-level protocols $\mathcal{P}_1, \mathcal{P}_2$, *etc*, which are encoded as state transition systems. All together, they form a part of the type environment when assigning a type to such a program. Thus, the Hoare type judgements assigning types to distributed

$$\dfrac{\mathcal{P}_1 \vdash e : \mathsf{DHT}\{\lambda s.\ A\}\big\{\lambda r\ s'.\ B\big\} \qquad A, B, R \text{ are stable} \qquad R \text{ constrains state related to } \mathcal{P}_2}{\mathcal{P}_1, \mathcal{P}_2 \vdash e : \mathsf{DHT}\{\lambda s.\ A \wedge R(s)\}\big\{\lambda r\ s'.\ B \wedge R(s')\big\}}\ \textsc{Inject}$$

$$\dfrac{\mathcal{P} \vdash e : \mathsf{DHT}\{\lambda s.\ A\}\big\{\lambda r\ s'.\ B\big\} \qquad I \text{ is inductive } wrt.\ \mathcal{P}}{\mathsf{WithInv}(\mathcal{P}, I) \vdash e : \mathsf{DHT}\{\lambda s.\ A \wedge I(s)\}\big\{\lambda r\ s'.\ B \wedge I(s')\big\}}\ \textsc{WithInv}$$

**Figure 1** Selected type inference rules of Distributed Hoare Types.

implementations are of the shape $\mathcal{P}_1, \ldots, \mathcal{P}_n \vdash e : \mathsf{DHT}\{\lambda s.\ A\}\{\lambda r\ s'.\ B\}$, where the typing context $\mathcal{P}_1, \ldots, \mathcal{P}_n$ lists all of the abstract protocols that the program $e$ can exercise, and the pre/postcondition constrain the state of the protocol-related part of the network. Each protocol defines the per-node local state, which is governed by the protocol's transitions. One node can possibly host disjoint pieces of local state that "belong" to different protocols, which is crucial to allow composing multiple protocols together to form useful systems. In addition to the send/receive primitives, all the standard programming constructs, such as conditionals, recursion, and higher-order functions can be used, and the typing rules for them are straightforward.

In any interesting distributed protocol, there are dependencies between messages about to be sent and the protocol-specific local state of a node that can send them. These dependencies are what our rich type system is designed to *enforce*. For instance, in any Paxos implementation, a replica can only send a response to a client when it is certain that agreement has been reached [61]. A protocol for Paxos would enforce this by constraining the *precondition* of sending a response to require that agreement had been reached. These constraints are manifested in the Hoare types, which are derived for the basic send/receive commands from the definitions of the transitions they follow. Since there is *no other way* to interact but by relying on the protocol-supplied transitions, this provides a powerful mechanism for enforcing system-specific constraints. For instance, in a well-typed program $e$, following a protocol $\mathcal{P}$, it will be only possible to send a certain message if the precondition in the corresponding transition $\tau_s$, is satisfied by the node's local state.[1]

The notion of well-typedness for Distributed Hoare Types incorporates program well-formedness with respect to the protocols in its typing context: no matter how complex the program is, if it is well-typed, then each of its externally observable transitions "faithfully" follows a transition of some of the protocols from its typing context, *i.e.*, it *does not go wrong* [38]. Summarizing the high-level overview of our approach, to enable language-based verification [53] of distributed systems, we have introduced the two following program- and type-level mechanisms to the otherwise well-studied model of higher-order effectful programs [50]:

**(a)** Instrumented message-passing primitives (send/receive), derived from protocol transitions, serving as basic building blocks for distributed programs;

**(b)** Distributed Hoare Types (DHT), an extension of Hoare Types [41], as a compositional approach to verify well-behaved programs in a context of arbitrary user-provided protocols.

**Addressing the Modularity Challenges.** Let us now see how the type-based approach helps alleviate the main difficulties of modular refinement proofs, outlined in Section 1.

[1] In fact, our type system allows for more general assertions, constraining the *global* state of the system.

1. Since any well-typed implementation must follow the protocol, type safety immediately implies refinement Moreover, Distributed Hoare Type Theory enjoys the standard substitution principle, which allows one to replace any program of a type $\mathsf{DHT}\{\lambda s.\ A\}\{\lambda r\ s'.\ B\}$ by any other program with the same type without compromising type safety.
2. From the perspective of a type system, there is no difference between a value obtained as a result of a local computation or the one received from a remote service, as long as it allows the desired Hoare type to be derived. Furthermore, Distributed Hoare Types allow for a form of *context weakening*, making it possible include more protocols (and account for interactions involving these protocols) into the typing context by adapting the pre/postconditions appropriately via the rule Inject from Figure 1.[2] The stability requirement on $R$ is standard for concurrency program logics and means that the assertions should be invariant with respect to possible concurrent changes in the network state [60].
3. The proof of an invariant $I$ being inductive with respect to a protocol $\mathcal{P}$ is not tied to a specific implementation $e$, and, therefore, can be discharged via an external verification tool (*e.g.*, Ivy [47]). That said, the invariant itself, once proven, can be used for strengthening the type of $e$, possibly enabling one to prove some properties of $e$'s clients. The interaction between protocol-level proofs and program-level verification is enabled by the typing rule WithInv from Figure 1. The *protocol combinator* WithInv enhances the state-space invariants of $\mathcal{P}$ conjoining them with the invariant $I$.

**Relation to Refinement Proofs.** Our careful choice of basic programming primitives, namely, *protocol transitions*, is the trick that allowed us to replace expensive proofs of program refinement with a far less complicated (although still non-decidable) and uniform type derivation mechanism. While this model might seem to be too "coarse-grained" in the sense that it forces changes in the protocol-relevant local state to be atomically synchronized with sending/receiving messages, the model nevertheless leaves a lot of room for possible program-level optimizations. Specifically, it allows one to combine the transitions in any well-typed way, as well as allowing one to make use of any internal state and higher-order programming primitives. What is more important is that our model explicitly identifies valid *linearization points* [17] in the implementations (they correspond precisely to the taken transitions), thus adopting a well-established proof method for observational refinement [14].

## 3 Language-Based Verification with Distributed Hoare Types

Distributed Hoare Types can be effectively represented as dependent types, parametrized by the protocol contexts and pre/postconditions [42]. This allowed us to implement the type-based verification approach, sketched in Section 2, in a verification tool Disel, by embedding our type system, its semantic foundations, and inference rules into the Coq proof assistant [6]. In this section, we outline the layout of specifications and proofs using a characteristic example of a widely-used distributed system: Two-Phase Commit (2PC) [63, §19].

The goal of the 2PC protocol is to achieve agreement among several nodes about whether a transaction should be committed or aborted (*e.g.*, as part of a distributed database). Since the system may execute in an asynchronous environment where message delivery is unreliable and machines may experience transient crashes, achieving agreement requires care. The protocol designates a single node as the *coordinator*, which is in charge of managing the

[2] These rules allow us to consider Distributed Hoare Types as a program logic-based verification framework.

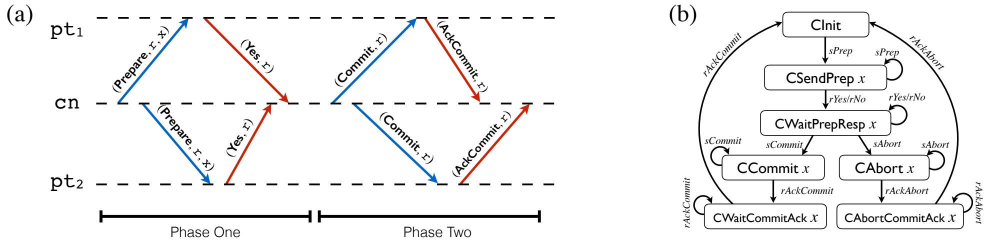



**Figure 2** One round of the 2PC algorithm (a) and state-space of the coordinator (b).

```
Definition c_recv_step (r : round) (cs : CState)
             (log : Log) (tag : nat) (mbody : seq nat) :=
  | CWaitPrepResp x ⇒ if (* received all votes *)
      then (r, if (* all votes yes *) then CCommit x else CAbort x, log)
      else (r, CWaitPrepResp, log)
  (* ... more cases depending on cs, tag, mbody ... *)
  end.
```

**Figure 3** Example receive transitions of the coordinator.

commit process; other nodes participating in the protocol are *participants*. The protocol proceeds in a series of rounds, each of which makes a single decision. Each round consists of two phases; an example round execution is shown in Figure 2(a). In phase one, the coordinator notifies the participants of the transaction being committed by sending *prepare* messages and receives *votes* from the participants about whether the transaction should proceed. In the figure, both participants vote Yes, so the coordinator enters phase two, during which it notifies all participants of its decision to commit or abort the transaction.

Formalizing this description into a protocol consists in describing the local state of each node as well as the valid transitions. Figure 2(b) shows the relevant portions of the local state of the coordinator and its transitions. Between rounds, the coordinator waits in the CInit state. Then, the coordinator makes transitions following the informal description above; these are formalized by the step-function, one case of which is shown in Figure 3. The additional state components keep track of the round number and a log of all processed transactions.

With the protocol instance in hand, we can now proceed to build programs that implement the participant and coordinator and assign them Hoare-style specifications. A possible implementation of a single round of the coordinator and its Hoare type are shown in Figure 4. The function `coordinator_round` takes as an argument the transaction data to be processed in this round. The type `DHT [cn, TPC]` is parametrized by the coordinator node id `cn` and a 2PC protocol instance `TPC`. The precondition requires that the coordinator is in the CInit state, with an arbitrary round number and log. The postcondition ensures that the local state has returned to CInit, the round number has been incremented, and the return value accurately reflects the decision made on the data, which is also reflected in the updated log. The code proceeds along the lines required by the protocol, but nothing prevents us from writing an optimized implementation, *adhering to the very same type*, which could, for instance, send abort-request upon receiving the first Phase One Abort response.

The type ascribed to `coordinator_round` above only constrains the local state of the coordinator, but in fact the protocol maintains stronger global invariants. For example, imagine using the Two-Phase Commit protocol as part of a larger distributed database system. Database nodes participate in several copies of the Two-Phase Commit protocol, one per node, so that each node is the coordinator of one copy of the protocol. Nodes

```
Program Definition coordinator_round (d : data) :
  {r log}, DHT [cn, TPC] (fun s ⇒ loc cn s = (r, CInit, log),
                          fun res s' ⇒ loc cn s' = (r+1, CInit, log ++ [(res, d)])) :=
  Do (r ← read_round;
      send_prep_loop r d;;
      res ← receive_prep_loop r;
      b ← read_resp_result;
      (if b then send_commits r d;;
                 receive_commit_loop r
       else send_aborts r d;;
            receive_abort_loop r);;
       return b).
```

**Figure 4** Distributed Hoare type and code of a single coordinator round.

can then commit transactions by initiating Two-Phase Commit in the copy of the protocol they coordinate. The database might like to conclude that between rounds, all logs are in agreement. This strong global agreement property is not directly implied by the protocol as it stands, so we must prove an inductive invariant that implies it. Finding such invariants typically requires several iterations before converging on a property that is inductive and implies the desired spec. In this case, a state invariant `Inv` that closely follows the intuitive execution of the protocol suffices to prove the global log agreement property. For example, when the coordinator is in the CSendCommit state, the invariant ensures that all participants are either waiting to hear about the decision, have received the decision but not acknowledged it, or have acknowledged the decision and returned to the initial state. The invariant also implies a simple statement of global log agreement, shown below.

```
Lemma cn_log_agreement (s : state) (r : round) (log : Log) :
  Inv s → loc cn s = (r, CInit, log) → ∀ pt, pt ∈ pts → loc pt s = (r, PInit, log).
```

In other words, when the coordinator `cn` is in the CInit state, all participants `pt` $\in$ `pts` must be in the PInit state with the same round number and log.

We can freely use the strengthened invariant in proofs of programs. For example, in the hypothetical database example, the programs implementing the database can now conclude global log agreement from the fact that the local state is CInit.

## 4 Related Work

### Type-based reasoning about concurrent and distributed systems

Session Types (ST) [18] are one of the most established approaches for lightweight verification of message-passing programs. ST were originally designed to constrain two-party channel communications, enforcing a particular interaction protocol; they were later extended to specify interactions between several parties [19, 8] and quantify over values of messages [55]. This has culminated in research on *choreographies* [3], which identify allowable orderings of message exchanges in a distributed system. Even though (Multiparty) Session Types (MST) [19] and Distributed Hoare Types pursue the same goal, namely, enforce the protocol discipline in an distributed setting with asynchronous message-passing, they seem to achieve this by different means. The underlying semantic formalism of MST is $\pi$-calculus [39], in which computations communicate via dedicated *session channels* that are a central notion for enforcing the well-formedness of executions via a tailored type system. In contrast, DHT adopts a model similar to those from modern program logics for fine-grained shared-memory concurrency [9, 40, 57, 54], in which messages of a specific protocol are treated as a *shared* *state*, related to local state of specific nodes via the protocol invariants and a subject to change as defined by the transitions.

While the precise relation between MST and DHT is still to be determined, we believe that our representation of distributed protocols via transition systems governing local/shared state is much closer in spirit to the models employed by the distributed systems community to describe the high-level logic of state-of-the art consensus and replication algorithms and their properties [25, 34]. It is not immediately clear to us how to encode Paxos, Raft or Two-Phase Commit using MST. Furthermore, the only language-level extension required to support a DHT programming model was the introduction of protocol-aware send/receive primitives and typing rules for them; the remaining language fragment is entirely standard. For instance, in our implementation the host language is Coq's Gallina [6] extended, via monadic embedding, with general recursion and message passing. This has the benefit that one can use the full power of Gallina to implement distributed programs. Finally, MST provide little support for reasoning about protocols themselves, separately from the programs they implement. This is something that is afforded for DHT using the WithInv rule.

A very close type-based formalism to DHT are RGRefs [15], allowing one to enforce a Rely/Guarantee-discipline [21] for mutable references in a shared-memory concurrency setting. That said, while RGRefs are suitable for showing that a program follows specific Rely/Guarantee-protocol, they are too weak to prove its invariants or functional correctness.

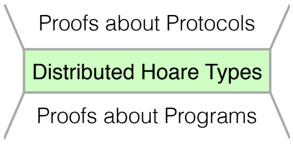
### Modular verification of distributed protocols

Compositional verification of invariants of distributed protocols is an area of active research in the Distributed Computing community (*cf.* [2, 33, 62]). There, it is common to reduce the reasoning about message-passing concurrency to reasoning about shared-memory mechanisms. For example, Boichat *et al.* [2] suggest a series of abstractions, such as *round-based consensus* and *round-based register*, that make it possible to deconstruct a family of Paxos algorithms into a set of reusable primitives. Input/Output automata [36] are another high-level formalism allowing for a form of protocol composition by coupling the automatas' actions [35]. At the moment, all these constructions are only studied at the level of reasoning about protocols, without any relation to implementations. We believe that these abstractions can be incorporated into the framework of DHT by generalizing the notion of the shared state to incorporate both message-passing (which is currently the case) and shared memory. Such a unification would make it possible to immediately reuse many of the existing specification and proof techniques from the logics for shared-memory concurrency, for instance, when defining custom correctness conditions [52].

Datta *et al.* [7] propose Protocol Composition Logic (PCL) as a way to combine security properties of multiple distributed protocols governing processes, communicating with each other. The programming component of PCL is a conventional process calculus. At the moment, it is not clear to us the extent to which PCL can be employed to verify, e.g., consensus protocols such as 2PC, or to be employed for reasoning about higher-order code.

## 5 Concluding Remarks

We have outlined the main ideas behind Distributed Hoare Types – a typing discipline that allows one to enforce high-level protocol logic in a low-level implementation via dependent types.




We believe that DHT serves as a link, connecting proofs of protocol properties and program properties in the same logical framework while providing modular reasoning. This modularization hints for a number of follow-up extensions, moving both up and down the abstraction stack.

**Moving up: Reasoning about protocols.** Thanks to the rule WithInv, reasoning about inductive protocol invariants can be conducted independently of the program-level verification. At the moment such proof obligations are discharged via Coq's native machinery for interactive proofs, and we are planning to investigate the possibility to delegate these proofs to third-party tools, such as Ivy [47], which is designed for this specific purpose. Furthermore, there is currently only one linguistic way to formulate protocols in the framework of DHT: by synchronizing the state changes with sending/receiving. This model is sufficiently fine-grained to be able to encode more transitional I/O Automata [35] or the round-based model [13] by establishing simulation *on the level of protocols* and generalizing the DHT semantics. Such a generalization is of practical interest, as it will allow us to port existing invariant proofs in other frameworks (*e.g.*, Verdi [64, 65]) that follow the I/O Automata model.

**Moving down: Reasoning about programs.** The immediate advantage of employing protocol-aware primitives for implementing provably correct distributed systems is the ability to use them in combination with higher-order functions and other programming mechanism. For instance, we were able to define loops and blocking receive just as syntactic sugar, relying on primitive commands and higher-order combinators. Even further, the shallow encoding of DHT into the Calculus of Constructions made it possible for use to take advantage of Coq's powerful abstraction mechanisms, providing reusable specifications for programs in terms of *abstract predicates* [9] rather then referring to concrete protocols. Finally, realistic distributed applications, such as multi-Paxos [61, 4] are far from being simple first-order code with message sending and receiving: they employ advanced features, such as per-node fork/join concurrency, higher-order iteration and client-side libraries. In order to establish the correctness of such implementations, one would have to relate the protocol-specific logic to those programming mechanisms – precisely what DHT enables.

That said, in the current formulation, the programming component of DHT is a pure functional language with general recursion and message passing. Imperative state and a form of exceptions can be encoded by means of "effect-passing" style, thus allowing some optimizations. For more low-level reasoning about highly optimized implementations in the presence of *native* mutable state, local faults, and per-node concurrency, we are planning to extend the reasoning with *low-level* versions of separation logic, adopting the ideas from the corresponding recent verification efforts [45, 5], as well as the idea of *transitions-as-resources* [29, 58] as a way to account for local concurrency, allowing several protocol branches to be exercised by a node in parallel [61].

## References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.

2 Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.

3 Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In *CONCUR*, pages 47–62. Springer, 2014.

4 Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407. ACM, 2007.

5 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37. ACM, 2015.

6 Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.5pl3*, 2016. `http://coq.inria.fr`.

7 Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

8 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011.

9 Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

10 Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. The Need for Language Support for Fault-Tolerant Distributed Systems. In *SNAPL*, volume 32 of *LIPIcs*, pages 90–102. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015.

11 Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415. ACM, 2016.

12 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15. ACM, 2009.

13 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

14 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

15 Colin S. Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, 2014.

16 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, pages 1–17. ACM, 2015.

17 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

18 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

20 Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, 2005. URL: `https://www.isa-afp.org/entries/DiskPaxos.shtml`.

21 Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

22 Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *PLDI*, pages 179–188. ACM, 2007.

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

**24** Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

**25** Leslie Lamport and Fred B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190 of *LNCS*, pages 203–285. Springer, 1985.

**26** Butler W. Lampson. How to build a highly available system using consensus. In *WDAG*, volume 1151 of *LNCS*, pages 1–17. Springer, 1996.

**27** Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370. ACM, 2016.

**28** Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574. ACM, 2013.

**29** Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.

**30** Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, pages 455–468, 2012.

**31** Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

**32** Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *OOPSLA*, pages 395–410, New York, NY, USA, 2012. ACM.

**33** Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. Brief announcement: A family of leaderless generalized-consensus algorithms. In *PODC*, pages 345–347. ACM, 2016.

**34** Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

**35** Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151. ACM, 1987.

**36** Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

**37** Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

**38** Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

**39** Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I, II. *Inf. Comput.*, 100(1):1–40, 1992.

**40** Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014.

**41** Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICFP*, pages 62–73. ACM, 2006.

**42** Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240. ACM Press, 2008.

**43** Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010.

**44** Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.

**45** Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. Fault-tolerant resource reasoning. In *APLAS*, volume 9458 of *LNCS*, pages 169–188. Springer, 2015.

**46** Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319. USENIX Association, 2014.

**47** Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630. ACM, 2016.

**48** Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. In *AVOCS*. EASST, 2015.

**49** John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

**50** Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.

**51** Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, volume 9032, pages 333–358. Springer, 2015.

**52** Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *OOPSLA*, pages 92–110. ACM, 2016.

**53** Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *FoSER*, pages 343–348. ACM, 2010.

**54** Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.

**55** Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP*, pages 161–172. ACM, 2011.

**56** Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390. ACM, 2013.

**57** Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.

**58** Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356. ACM, 2013.

**59** Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL*, pages 247–258. ACM, 2011.

**60** Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, volume 4703, pages 256–271. Springer, 2007.

**61** Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, 2015.

**62** Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Trans. Dependable Sec. Comput.*, 12(4):472–484, 2015.

**63** Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

**64** James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368. ACM, 2015.

**65** Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for change in a formal verification of the Raft Consensus Protocol. In *CPP*, pages 154–165. ACM, 2016.