FMitF: Track I: ADVERT: Compositional Atomic Specifications for Distributed System Verification

Zhong Shao (PI)
Robert Soulé (co-PI)
Ji Yong Shin (co-PI)
Department of Computer Science
Yale University
51 Prospect Street
New Haven, CT 06520-8285, USA
Phone: 203-432-6828
Email: {zhong.shao, robert.soule, jiyong.shin}@yale.edu

A proposal for Computing and Communication Foundations (CCF): Formal Methods in the Field (FMitF)
NSF program solicitation 19-613

Project Summary

Overview: Distributed systems are difficult to verify due to their inherent complexity, which stems from handling concurrency and network asynchrony. Significant advances have been made in formally reasoning and verifying distributed systems, but most existing approaches focus on reasoning about specific instances of distributed systems and do little to explore how to expose the common high-level behaviors of distributed systems while hiding the implementation details. Therefore, verifying individual distributed systems requires redundant reasoning and the absence of a high-level model makes it difficult to address the new challenges that modern applications are often composed of multiple distributed systems. Most current verification tools and approaches are not well-equipped to handle multiple distributed system interactions and the PIs aim to address this problem by proposing a high-level uniform model of a distributed system that facilitates reasoning and verification of both individual and composition of distributed systems.

Keywords: distributed/operating systems; atomic distributed system specification; distributed system verification; verified distributed system composition; formal specification; formal verification.

Intellectual Merit: In this effort, the PIs propose to design a novel compositional atomic distributed object model that can be used to reduce the complexity of distributed system reasoning and verification. The atomic distributed object model is defined using modified push/pull operations from work on shared-memory concurrency and maintains a logical history of state change requests. The model hides details such as interleaved network messages and failures that complicates the reasoning about the core safety properties of a distributed system. Still, the model provides enough detail to reason about a system at a much higher abstraction level.

The atomic distributed object model encapsulates the key safety properties of individual distributed systems and the PIs plan to build multiple network-based specification that captures the common network-level behavior of similar classes of distributed systems. The network-based specification helps individual systems to verify their refinement relation to the atomic distributed object model, provides reusable proofs that are derived from common system behaviors, and acts as a verification template that verifies the safety properties encapsulated in the atomic object model for free.

Once individual distributed systems are verified to be correct and safe based on the atomic distributed object model, the high-level abstraction of the model can be used to reason about multiple distributed system interactions. Verification of multiple distributed system interactions is challenging to do with existing approaches, which lack a common high-level model of individual systems. The atomic distributed object model can play a key role to stitch multiple verified distributed systems together into a larger distributed object without having to deal with the low level details of individual systems.

The PIs will build in Coq a distributed system verification framework, ADVERT, based around the atomic distributed object model to verify individual distributed systems and their interactions. The PIs will demonstrate through concrete examples that proving properties even of composite distributed systems can be straightforward with the atomic distributed object model due to the elegantly simple object interface. The PIs plan to verify real-world cutting-edge distributed systems written in C. One of the target systems is a distributed shared memory that uses a programmable switch, a low-latency network, and multiple sharded distributed components that run consensus protocols.

Broader Impact: The technology for simplifying distributed system reasoning and allowing for verification of distributed system composition will have a profound impact on the software industry and the society in general. It will dramatically improve the reliability and security of large-scale software infrastructures, such as the cloud, and applications that run on top of the infrastructure. The atomic distributed object specification and the verification framework will make distributed infrastructures easier to understand and verify. The applicability of the research outcome can be easily extended to relevant fields such as cyber physical systems or internet of things that use multiple sensors and devices over the network. On the educational side, this project will push new courses on distributed system design and verification and will broaden the participation of underrepresented groups.

1 Introduction

Distributed systems are difficult to reason about and verify because they employ sophisticated protocols to coordinate failure-prone distributed nodes over an unreliable network. Their correctness often relies on subtle, implicit global invariants, which hinder the ability to understand why the system works and how one system relates to another. To further complicate matters, modern applications build on top of multiple distributed systems [1, 18], which necessitates reasoning about concurrent interactions between systems.

Formal models and verification of the code are necessary to guarantee the safety and correctness of a distributed system. However, it is challenging to verify a standalone distributed system and harder still to verify multiple interacting distributed systems [5]. There exist amazing efforts to apply formal reasoning and verification to distributed systems that provide useful tools for simplifying some aspects of verification, such as high-level state machine refinement methodology [9], network model transformation [30], automatic invariant generation [20], and modular reasoning for multiple distributed protocols with simple interactions [25]. However, they work at a relatively low level while dealing with implementation details, so verifying individual distributed systems requires redundant reasoning. Similarly, the low-level details make it difficult to address the new challenges that come with composition of systems.

Then how can we facilitate the verification of multiple individual distributed systems and their compositions? We claim that providing a simple, atomic, and general interface is key to reducing the complexity of verifying distributed systems. It is well known that providing a simple object model by abstracting away implementation details can greatly ease the understanding and verification of complex systems [11, 10, 26, 8]. While existing efforts listed above do employ some amount of abstraction, non-essential details, such as message orderings and quorum sizes, sometimes leak through into the high-level reasoning. Thus, by modeling distributed systems at an even higher level, one can focus on just the core logic, and deal with the implementation details separately.

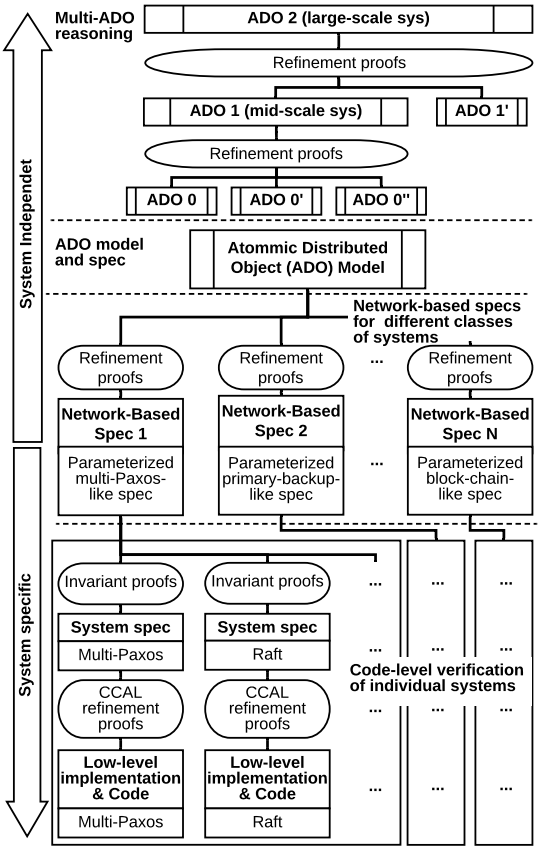

Figure 1: ADVERT blueprint: four major components, including the atomic distributed object specification, and proofs that bridge the components.

We propose to design a new high-level abstraction of distributed systems to justify our claim. We aim to realize a specification and verification framework, ADVERT, for individual distributed systems and their composition surrounding the abstraction, and model and verify distributed systems down to the code level. ADVERT stands for atomic distributed object verification toolchain and targets a general verification approach for strongly consistent replicated distributed systems that satisfy the replicated state safety property (see Theorem 1). Examples of such systems include multi-Paxos [28], Raft [24], and chain-replication [29], which are widely used in production systems. ADVERT aims to model and verify composition of such systems as long as the composition satisfies the replicated safety property.

Inspired by work on shared-memory concurrent objects [10, 2], we propose a novel model of distributed systems called the atomic distributed object (ADO) and plan to build ADVERT surrounding the ADO. The ADO models all distributed operations as atomic functions on a non-distributed shared object, but with added

non-determinism to reflect its distributed nature. The calls are captured by atomic updates to a linear, logical history of events. The ADO model is detailed enough to express high-level behaviors of distributed systems, but abstract enough that implementation details do not unnecessarily complicate the high-level reasoning.

We plan to divide ADVERT into four major components (Figure 1) and implement each component in Coq [27]: 1) the ADO specification, 2) a multi-ADO reasoning framework, 3) a network-based specification, and 4) a code-level verification framework. The ADO specification sits at the center of ADVERT to model individual distributed systems using a simple atomic interface. The multi-ADO reasoning framework allows composite systems to be built from ADOs and reasoned about modularly. ADVERT fills in the missing details from below with the network-based specification, and provides a clear path between implementation and ADO by proving a refinement relation. This specification will be parameterized such that it can model most protocols with similar network patterns. Along the way we will solve the challenges that stem from mapping arbitrarily delayed future events in the network-level specification to an atomic function call in the ADO model. Verifying that different systems refine the ADO model will be made simple and convenient through the network-based specification because the properties captured by the common network pattern could allow for the reuse of proofs. Finally, the code level verification framework will use the certified concurrent abstraction layer (CCAL) [8] approach (the PI's prior work). ADVERT will take care of each step to connect the code-level implementation of a system up to multiple distributed system reasoning.

Our proposed research consists of the following four components:

- We will identify and propose an atomic distributed object model that preserves the key characteristics of distributed systems, but hides the implementation details.
- We will explore different distributed protocols to create parameterized network-based specifications that connect individual system implementations to the atomic distributed object model through a refinement relation. Each network-based specification will reuse the refinement proofs and act as a template that connects a similar class of individual systems to the atomic distributed object model.
- We will conduct studies of composite distributed applications to illustrate how ADVERT and atomic distributed object model enable simple reasoning about multiple distributed system interactions. We will explore various patterns of system compositions for broad impact.
- To demonstrate the power and real-world applicability of ADVERT, we will carry out concrete instantiations multiple distributed protocols, including multi-Paxos, and Raft, and extend the instantiations to a large-scale distributed shared memory system.

Related Efforts The PI is a co-investigator of the NSF-funded DeepSpec/CertiKOS project. The ADVERT project differs from the CertiKOS project as it focuses on distributed systems which run above operating system layers. CertiKOS tackles challenges of verifying operating system code base and handling concurrency within operating system. The ADVERT project is different from the CertiKOS project as it reasons about distributed system concurrency, where completely different failure and network conditions are present. The ADVERT project rather complements the CertiKOS project by extending the verified software stack up to a composition of distributed applications.

Other efforts to facilitate the verification of distributed systems are introduced at the start of this section [9, 30, 20, 25]. All these efforts do little to explore how to expose the common high-level behaviors of distributed systems and hide the implementation details. They do not explore reusable proofs to verify multiple distinct distributed systems using a common framework and do not focus on verifying compositions of multiple distributed systems. Our grand goal is to realize a distributed system modeling and verification framework that can verify both standalone and composite distributed systems to create a fully verified distributed ecosystem.

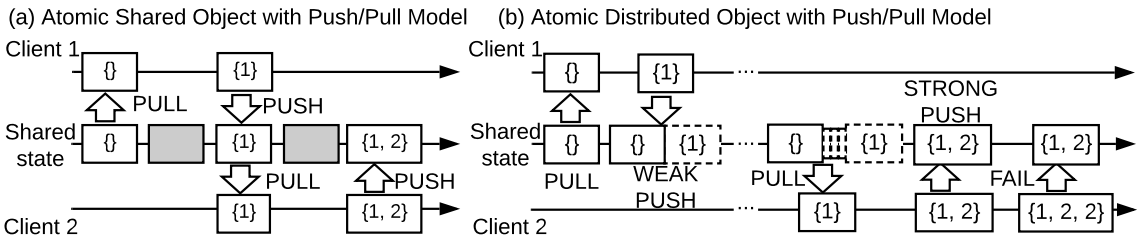

Figure 2: Shared queue examples that compare push/pull for atomic shared and atomic distributed objects.

2 Atomic Distributed Object Model

ADVERT models distributed systems as atomic distributed objects (ADO). An ADO contains some internal state and provides a set of atomic primitives that clients can invoke to access or modify the state. These primitives can then be combined to build more elaborate operations called methods, which act as the interface through which ADOs can interact. An important property of ADOs is that regardless of the underlying implementation or how many clients access an object concurrently, the semantics of the atomic primitives guarantee that the object's state evolves according to a sequential history of atomic updates. The ADO model is therefore a powerful abstraction because it accurately captures the high-level essence of distributed system behavior while remaining oblivious to the details of the complex protocols running under the hood. The ADO model becomes a tool to capture key common properties of many different distributed systems so that proofs for the properties can be reused and composition of multipe systems can be easily verified. We present our preliminary model of ADO and its power which will be iteratively revised as our research makes progress.

2.1 Shared Memory Analogy

Before we present the formal details of the atomic distributed object model, we provide an intuition for it through an informal analogy to a push/pull concurrent shared memory model [15, 13, 7, 8].

Push/Pull Basics Figure 2 (a) shows an example of an atomic object implemented in a (non-distributed) shared memory using push/pull operations. It consists of a single shared queue and two clients (threads) that attempt to concurrently enqueue values.

When client 1 wishes to update the shared state, it first *pulls* a copy of the shared queue into a local cache. It finds that it is empty and then enqueues 1 into its local copy. Next, to announce this change globally, client 1 *pushes* its modified copy, which flushes the cache back to the shared memory.

The example becomes more interesting when we consider multiple clients attempting to push and pull concurrently. To maintain atomicity and avoid data races, at most one client at a time should "own" the state by pulling it. This is represented in the figure by drawing the state as a gray box while a client has pulled a copy. Attempting another pull at such times is undefined behavior. Therefore, client 2 must wait until client 1 pushes the update back before it can pull and enqueue its own value.

Mapping to Distributed Systems The push/pull model provides an effective abstraction for building and reasoning about atomic concurrent objects, but it does not fully capture distributed system behaviors. For example, network errors and node failures can cause updates to fail, or only partially succeed. Additionally, distributed systems typically do not provide mutual exclusion and ownership of the shared state can be preempted. We now present an extended push/pull model that addresses these shortcomings.

As before, there is a single shared state that can be accessed concurrently by clients (distributed nodes); however, the state now consists of a persistent component and a set of shared volatile caches. The caches

$$
\begin{array}{lcll}
\mathsf{RDATA} & : & Type & \text{(Replicated State)} \\
\mathsf{EV_{RDATA}} & := & (\mathbb{Z} * \mathsf{RDATA}) \to \mathsf{RDATA} & \text{(Event)} \\
\mathsf{H_{RDATA}} & := & \text{list } (\mathbb{Z} * \mathsf{EV_{RDATA}}) & \text{(History)} \\
\mathbb{R}_{\mathsf{RDATA}} & : & \mathsf{H_{RDATA}} \to \mathsf{RDATA} & \text{(Replay)} \\
\mathsf{DSM} & := & \{smem : \mathsf{H_{RDATA}}; & \\
 & & \ caches : \{\mathbb{Z} \mapsto \mathsf{H_{RDATA}}\}; & \\
 & & \ npull : \{\mathbb{Z} \mapsto \mathbb{Z}\}; & \\
 & & \ owner : \mathbb{Z};\ cacheID : \mathbb{Z}\} & \text{(State)}
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{RES_{pull}} & := & \mathsf{PULL}\ (cID : \mathbb{Z}) \mid \mathsf{FAIL} & \\
\mathsf{RES_{push}} & := & \mathsf{SPUSH}\ (res : \mathbb{B}) \mid \mathsf{WPUSH} & \\
 & & \mid \mathsf{FAIL} & \\
\mathbb{O}_{\text{pull}} & : & \mathbb{Z} \to \mathbb{Z} \to \mathsf{RES_{pull}} & \text{(Pull Oracle)} \\
\mathbb{O}_{\text{push}} & : & \mathbb{Z} \to \mathbb{Z} \to \mathsf{RES_{push}} & \text{(Push Oracle)}
\end{array}
$$

PULLSUCCESS

$$\frac{\begin{array}{c} st, st' \in \mathsf{DSM} \qquad cID \in \mathbb{Z} \qquad npull_{id} = st.npull(id) \\ \mathbb{O}_{\text{pull}}\ id\ npull_{id} = \mathsf{PULL}\ cID \qquad npull' = st.npull/[id := npull_{id} + 1] \qquad st' = \{st.smem, st.caches, npull', id, cID\} \end{array}}{\mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{pull}}(id : \mathbb{Z}) : st \to (st', \mathsf{true})}$$

PULLFAIL

$$\frac{\begin{array}{c} st, st' \in \mathsf{DSM} \qquad npull_{id} = st.npull(id) \qquad \mathbb{O}_{\text{pull}}\ id\ npull_{id} = \mathsf{FAIL} \\ npull' = st.npull/[id := npull_{id} + 1] \qquad st' = \{st.smem, st.caches, npull', \bot, st.cacheID\} \end{array}}{\mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{pull}}(id : \mathbb{Z}) : st \to (st', \mathsf{false})}$$

STRONGPUSH

$$\frac{\begin{array}{c} st, st' \in \mathsf{DSM} \qquad st.owner = id \qquad npull_{id} = st.npull(id) \qquad \mathbb{O}_{\text{push}}\ id\ npull_{id} = \mathsf{SPUSH}\ res \\ smem' = st.smem \bullet st.caches(st.cacheID) \bullet (id, ev) \qquad st' = \{smem', \emptyset, st.npull, \bot, st.cacheID\} \end{array}}{\mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{push}}(id : \mathbb{Z}, ev : \mathsf{EV_{RDATA}}) : st \to (st', res)}$$

WEAKPUSH

$$\frac{\begin{array}{c} st, st' \in \mathsf{DSM} \qquad st.owner = id \qquad npull_{id} = st.npull(id) \\ \mathbb{O}_{\text{push}}\ id\ npull_{id} = \mathsf{WPUSH} \qquad cache_{id} = st.caches(st.cacheID) \bullet (id, ev) \qquad st.caches(freshID) = \mathsf{nil} \\ caches' = st.caches/[freshID := cache_{id}] \qquad st' = \{st.smem, caches', st.npull, \bot, st.cacheID\} \end{array}}{\mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{push}}(id : \mathbb{Z}, ev : \mathsf{EV_{RDATA}}) : st \to (st', \mathsf{false})}$$

PUSHFAIL

$$\frac{\begin{array}{c} st, st' \in \mathsf{DSM} \qquad st.owner = id \\ npull_{id} = st.npull(id) \qquad \mathbb{O}_{\text{push}}\ id\ npull_{id} = \mathsf{FAIL} \qquad st' = \{st.smem, st.caches, st.npull, \bot, st.cacheID\} \end{array}}{\mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{push}}(id : \mathbb{Z}, ev : \mathsf{EV_{RDATA}}) : st \to (st', \mathsf{false})}$$

Figure 3: Atomic Distributed Object Semantics

represent data that is not yet replicated across enough servers and may be dropped or overwritten. Figure 2 (b) shows atomic operations on a distributed queue similar to the previous example. As before, client 1 begins by pulling the shared state, but note that its ownership is no longer denoted by a gray box, which indicates that other clients are allowed preempt it. After enqueueing 1 it pushes its update, but it results in a *weak push*, which leaves the persistent state untouched and creates a new cache. When client 2 later pulls the state it non-deterministically chooses between the persistent memory and one of the caches. It selects the cache with client 1's update, modifies it locally, and then pushes it back. This time the result is a *strong push*, which succeeds in updating the persistent state and clears all other caches. It is also possible for a push to fail completely, leaving the shared state unchanged (e.g., the result of client 2's final push attempt).

The push/pull behavior mirrors non-deterministic distributed system behaviors from network interleavings and failures. Multiple cache states models partially committed and thus inconsistent states and the persistent memory maps to consistently committed state. The ADO model captures how a standalone distributed system behaves and its high-level APIs hides sophisticated distributed system decisions so that reasoning about multiple distributed system interactions can be simplified. In fact, the behavior of the queue in Figure 2 appears very much like a high-level view of a state machine replication protocol like multi-Paxos.

2.2 Atomic Distributed Object Specification

The formal semantics and relevant types of the ADO model using the extended push/pull model are sketched in Figure 3. σ_x denotes the specification of the function x, and • is list concatenation.

The type of the replicated shared state (RDATA) is a parameter of an object, but rather than maintaining a concrete value, we record a logical history ($\mathsf{H}_{\mathsf{RDATA}}$) of events ($\mathsf{EV}_{\mathsf{RDATA}}$), which facilitates reasoning about how the object evolves over time. An event is a function from a client identifier and RDATA pair to a new RDATA, so the concrete state can be computed at any point by applying a *replay function* $\mathbb{R}_{\mathsf{RDATA}}$ to a history, which simply applies each event in turn.

An ADO contains a data structure called a DSM (for Distributed Shared Memory) that stores the shared state and associated metadata. The five components are: 1) $smem$, the persistent component of the shared state represented as a history; 2) $caches$, a set of histories for events that are not fully replicated; 3) $npull$, a set of the number of times each client has pulled the shared state; 4) $owner$, the ID of the client that last pulled successfully; and 5) $cacheID$, the ID of the cache that the current owner pulled.

For a concrete example of how an atomic distributed object looks, refer to Figure 4, which presents the distributed queue from Figure 2 (b) in the ADO style. The concrete state RDATA is instantiated as a pair of an array and the index of the queue's tail. The only event it can use to modify the queue is `enq`, which enqueues the client's ID. The interface it exposes consists of two methods: `Enqueue`, which is simply a wrapper around `enq` that performs the necessary pull and push operations; and `Tail`, which uses the replay function to compute the queue's tail. Note that methods implicitly check for failure after push and pull and return early if so.

```
ADO Queue:
  DSM[{tl: ℤ; vals: {ℤ ↦ ℤ}}] queue
  Event enq (id, q) =
    {q.tl + 1, q.vals/[q.tl := id]}
  Method Enqueue = this.queue := pull;
    this.queue := push enq
  Method Tail =
    this.queue := pull;
    q := replay this.queue;
    return q.vals[q.tl − 1]
```

Figure 4: A distributed queue modeled as an ADO.

Pull and push define transitions from one DSM to a new one. They also return a boolean value indicating whether the client that performed the operation observed it as a success or failure, which is always an underapproximation of the true result. This captures the case in a distributed system where a client's request was committed, but the acknowledgement messages were lost. Pull and push must non-deterministically choose between success and failure cases, so their specifications are parameterized by abstract *oracles* ($\mathbb{O}_{\text{pull}}$ and $\mathbb{O}_{\text{push}}$) that make these decisions. These are functions that, given a client ID and $npull$, arbitrarily determine the result of the push or pull operation. Modeling the non-determinism in this way simplifies relating the ADO model to the network-based specification, which also relies on oracles (Section 3). Pull has two outcomes: success (PULL) and failure (FAIL). A successful pull increments the pull count, makes the client the owner, and sets the current cache ID to the one arbitrarily chosen by $\mathbb{O}_{\text{pull}}$. In a failed pull, although the client does not become the owner, it invalidates the previous ownership by setting the owner to $\perp$.

Push has three outcomes: strong push (SPUSH), weak push (WPUSH), and failure (FAIL). Each case requires that the client is the current owner and afterwards sets the owner to $\perp$, which implies a push must follow a successful pull [1]. A strong push flushes the current cache to the persistent state, appends the new event, and clears all other caches. A weak push leaves the persistent state and existing caches unchanged, but creates a new cache by appending the new event to the current cache. A failed push does not change the state except for clearing the owner.

The ADO model is general enough to capture the behaviors of many distributed systems. It is interesting then to examine what properties it satisfies because they must also hold for any system that implements the

[1] This requirement can be relaxed somewhat by allowing strong pushes to retain ownership.

$$\begin{array}{lcll} \mathsf{LDSM} & := & \{owns : \mathbb{B}; id : \mathbb{Z};\ snap : \mathsf{DSM}\} & \text{(Local State)} \\ \varepsilon_{\mathsf{ADO}} & : & \mathbb{Z} \to \mathsf{DSM} \to \mathsf{DSM} & \end{array}$$

LOCALPULLSUCCESS

$$\frac{lst, lst' \in \mathsf{LDSM} \quad snap' = \varepsilon_{\mathsf{ADO}}\ lst.id\ lst.snap \quad \mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{pull}}\ lst.id\ snap' = (snap'', \mathsf{true}) \quad lst' = \{\mathsf{true}, lst.id, snap''\} \quad rdata = \mathbb{R}_{\mathsf{RDATA}}\ (snap''.smem \bullet snap''.caches(snap''.cacheID))}{\varepsilon_{\mathsf{ADO}}, \mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{pull}_{\text{local}}} : lst \to (lst', rdata)}$$

LOCALPULLFAIL

$$\frac{lst, lst' \in \mathsf{LDSM} \quad snap' = \varepsilon_{\mathsf{ADO}}\ lst.id\ lst.snap \quad \mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{pull}}\ lst.id\ snap' = (snap'', \mathsf{false}) \quad lst' = \{\mathsf{false}, lst.id, snap''\}}{\varepsilon_{\mathsf{ADO}}, \mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{pull}_{\text{local}}} : lst \to (lst', \bot)}$$

LOCALPUSH

$$\frac{lst, lst' \in \mathsf{LDSM} \quad lst.owns = \mathsf{true} \quad \mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{push}}\ ev\ lst.id\ lst.snap = (snap'', res) \quad lst' = \{res, lst.id, snap''\}}{\varepsilon_{\mathsf{ADO}}, \mathbb{O}_{\text{pull}}, \mathbb{O}_{\text{push}} \vdash \sigma_{\text{push}_{\text{local}}}(ev : \mathsf{EV}_{\mathsf{RDATA}}) : lst \to (lst', res)}$$

Figure 5: Local Atomic Distributed Object Semantics

ADO specification. One such property is the important safety condition that the persistent shared state is truly persistent (Theorem 1).

Theorem 1 (Replicated State Safety)**.** *For all $st \in \mathsf{DSM}$, let st' be the result of arbitrarily many applications of σ_{pull} and σ_{push}. Then the following are true:*

1. $\exists id.\ st.npull(id) + 1 = st'.npull(id) \wedge (\forall id'.\ id' \neq id \to st.npull(nid) = st'.npull(nid))$
2. $st.smem = \textbf{\textit{prefix}}\ st'.smem$

The proof comes essentially for free from the push/pull semantics since every rule can only grow $smem$ and only overwrites $caches$ when $smem$ grows. This shows that reasoning at the level of the ADO model can be quite straightforward despite the complexity of the underlying distributed system that implements it.

2.3 Local Atomic Distributed Object View

The ADO model captures the "global" view of an object, but a client of the object has a different view. The global view distinguishes each operations by the client that invoked it, but to a client's local view, it knows only its own operations and everything else is due to the external environment. Using a local view allows one to reason in a more modular and compositional manner [8, 14, 19], so we define local versions of push and pull for the ADO model as well. This is done with the help of an *environment context* ($\varepsilon_{\mathsf{ADO}}$), which is responsible for representing the actions of all other clients.

Figure 5 contains the formal definitions of $\sigma_{\text{pull}_{\text{local}}}$ and $\sigma_{\text{push}_{\text{local}}}$. The local state (LDSM) consists of three fields: a flag ($owns$) indicating whether the client believes it is the owner, a client identifier (id), and a snapshot of the shared state ($snap$). The environment context $\varepsilon_{\mathsf{ADO}}$ is a function that takes a client ID and current snapshot, and returns a new snapshot that is the result of all the operations performed by other clients.

The specifications for the local versions of push and pull are simple wrappers around the global versions. Pull uses the environment context to retrieve an up-to-date snapshot and then calls σ_{pull}. If pull is successful it replays the history stored in its snapshot and returns the computed concrete state along with the new LDSM. If we assume that $\varepsilon_{\mathsf{ADO}}$ only returns DSMs that can be reached by the global push and pull specifications, then because these specifications are defined in terms of the global versions, it is clear that properties that hold in the global specification, such as Theorem 1, also hold here.

2.4 ADO Composition

Reasoning about distributed systems with an implementation-level specification can be very complicated because the application logic becomes entangled with the underlying distributed protocol logic. One of the major benefits of modeling distributed systems as ADOs is that their behavior can be understood purely at the level of push and pull opertions without any knowledge of the underlying network model. This is especially useful when considering a larger system that consists of multiple ADOs, because the composite system's behavior can be understood in terms of its components' atomic specifications.

Given two ADOs X and Y, their horizontal composition ($X \oplus Y$) is a new ADO that takes the product of their replicated state and merges their histories. More precisely, if $Z = X \oplus Y$ then $\mathsf{RDATA}_Z = \mathsf{RDATA}_X * \mathsf{RDATA}_Y$. Events for X can be trivially lifted to events for Z by the function $lift_l = \lambda ev_X\ st_Z.(ev_X\ (fst\ st_Z), snd\ st_Z)$ and similarly for Y. Methods can also be automatically lifted by lifting events and applying projections to RDATA as necessary. It is clear then from these definitions that Z simply represents X and Y running independently side-by-side.

Z can be made more interesting by defining methods that introduce some dependency between X and Y. For example, suppose Z uses X to store data and Y to maintain related metadata; then Z must ensure that X and Y remain synchronized. This type of horizontal composition of distributed systems with disjoint state is also supported by DISEL [25], albeit at a lower abstraction level. DISEL does not, however, address vertical composition, in which objects are transitively refined into increasingly abstract specifications. In the ADO model this is as simple as defining a refinement relation between two objects' RDATAs and proving that their methods preserve the relation. By repeatedly composing and refining ADOs in this way, one can build large applications out of modular components that can be reasoned about independently.

3 Connecting to Network-Based Specifications

The ADO model is convenient for high-level reasoning because it hides many of the less-important details of distributed systems such as network communications, quorum sizes, etc. Unfortunately, in doing so it also obscures its relation to concrete implementations, which makes it more challenging to prove that it accurately describes the behavior of a particular protocol.

To address this problem we plan to create a lower-level specification based on network-level behaviors, which can act as an intermediate step in a refinement proof. To take advantage of the fact that many distributed protocols essentially differ only in a few details, the network-based specification is actually a family of specifications where certain values and functions are left as parameters. For example, the specification that captures Paxos-like behaviors can model Paxos variants such multi-Paxos [28], Raft [24], and Mencius [21].

3.1 Network Model

Distributed systems have different assumptions about the network. For example, Paxos assumes non-Byzantine asynchronous network, where messages can be arbitrarily reordered, delayed, lost, and duplicated but are never corrupted, whereas Zab [12] assumes ordered message delivery and relies on the TCP protocol.

Research Task 1a: Flexible Network Model Design In order to model and verify various distributed systems with ADO and ADVERT, we will create a network model that can accommodate different network assumptions. Similar to how a real network is designed, we will start from a very weak network and then build more constraints on top of it. We intend to parameterize the network model so that the network semantics can be configured to satisfy the system needs.

The network can be modeled by maintaining a log of network events parameterized by an arbitrary message type (Figure 6). The five network events are send (Snd), receive (Rcv), timeout (TO), broadcasting

send ($Bsnd$), and ghost ($Ghst$). Snd and Rcv represent the moment when a node successfully sends or receives a message, and they carry the (unique) node IDs of the source and destination along with a message. TO represents the case when a node finds it has no received messages, and $Bsnd$ is a convenient abbreviation for a sequence of sends to every node in the system. Unlike the others, $Ghst$ events do not correspond to any real network communications, but are purely logical constructions that can be useful for reasoning about network patterns. For example, a client can use them to mark the beginning and end of an update operation, which then provide convenient boundaries for relating sequences of network events to atomic operations.

The network log ($Log_{\texttt{net}}$) represents a linearized view of the history of all network events, which reflects the fact that although events may occur concurrently, they are observed by a node in some sequential order imposed by the hardware and software (e.g., the network card and operating system). In order for our proofs to remain agnostic to any particular ordering we appeal to an abstract function called the network oracle ($\mathbb{O}_{\text{net}}$). The oracle takes as input the current network log, and returns a list of the events produced by the external world, thereby choosing an arbitrary ordering. $\text{valid}_{\texttt{net}}$ is a user-defined invariant on network logs, such as requiring that all send events originate from a certain set of nodes.

$$
\begin{array}{lcl}
msg\_t & : & Type \\
gmsg\_t & : & Type \\
nid\_t & := & \mathbb{Z} \\
E_{\texttt{net}} & := & Snd(nid\_t * nid\_t * msg\_t) \\
 & & \mid Rcv(nid\_t * nid\_t * msg\_t) \\
 & & \mid TO(nid\_t) \\
 & & \mid Bsnd(nid\_t * msg\_t) \\
 & & \mid Ghst(nid\_t * gmsg\_t) \\
Log_{\texttt{net}} & := & \text{list } E_{\texttt{net}} \\
\mathbb{O}_{\text{net}} & : & nid\_t \to Log_{\texttt{net}} \to Log_{\texttt{net}} \\
\text{valid}_{\texttt{net}} & : & Log_{\texttt{net}} \to \text{Prop}
\end{array}
$$

Figure 6: Network definitions

Based on the network type and the network oracle, we plan to parameterize the network semantics such that the network behavior can be configured for different needs. We will start from the non-Byzantine asynchronous network and add constraints for example to enforce ordering or guaranteed delivery of messages.

3.2 Network-Based Specification

When equipped with a parameterized network model, we can instantiate the model and write network-based specifications of different systems. We plan to divide the network-based specification into three parts: 1) a network replay function, which computes node states from a network log; 2) client and server specifications, which build on the replay function and preserve invariants about network patterns; and 3) parameterized protocol-specific components, which enable the specification to describe multiple distributed protocols.

Research Task 1b: Network Replay Function The key idea to build the network replay function is that state transitions are triggered by network events, so the network log contains enough information to compute every node's state. Surrounding the network log ($Log_{\texttt{net}}$), we plan to build the replay function to reconstruct and reason about the entire system state. The log and the replay function will essentially act similar to a state machine and reveal each step of a distributed protocol and interleaving of concurrent nodes, so that all logically possible behaviors of a distributed operation can be verified.

Research Task 1c: Client and Server Specifications Whereas the replay function defines how a network log is consumed, the functions within the network replay function that transform network events into new network events and concrete system states are specific to a class of distributed systems. For example, Paxos, Raft, and Paxos variants share similar two-phase communication patterns and maintain similar data states (e.g., current logical time as round numbers, log of data, time stamps of data writes, etc.), so they can be specified under similar high-level specifications. To model various systems into ADO through the network-based specification, we plan to categorize many different systems into few representative classes so that widely used patterns can be reused.

Research Task 1d: Protocol-Specific Parameters Even though the client and server specifications model representative classes of distributed systems at a high-level, low-level details of a system should be mapped to the specification to fully model a specific system. Therefore, we plan to leave certain elements of the specifications to be parameterized. By not fixing these components the specification will become more flexible and capable of modeling a wider range of implementations.

The parameterization requires careful study of multiple systems. We believe there is a trade off between how much flexible the specification can be and how sophisticated it becomes to connect the specification to the ADO model through refinement proofs. In addition, the degree of flexibility in each specification can affect the number of distributed system classes that we model: for example, a very flexible specification design may cover 20 system instances but the refinement proof may become difficult, whereas a rigid specification may cover only 5 instances but the proof becomes easy. We aim to find the sweet spot in this space.

3.3 Connections Between ADO and Network-Based Specifications

The ADO specification is useful for reasoning about distributed systems, but it does not match their low-level implementations very closely. The network-based specification on the other hand maps more directly onto the implementations, but contains too many implementation details for simple reasoning. Proving a refinement between the two specifications allows one to take advantage of the strengths of both.

One can prove an implementation correct against the network-based specification, and then prove properties at the LDSM level that are propagated down through the refinement relation. For the refinement to be useful, it is important to define a sensible refinement relation that is strong enough to capture the desired properties but weak enough to be provable; however, this turns out to be surprisingly difficult for these specifications due to differences in when they consider operations to be complete. We first discuss the considerations and challenges in defining a refinement relation.

Problem Definition $\mathbb{R}_{\mathsf{LDSM}}$ is a relation between a client's LDSM and a $Log_{\texttt{net}}$. The property it must guarantee is that the shared state in both models is observably equivalent. The DSM snapshot is the interesting part of the LDSM, so the following discussion focuses primarily on that. In the ADO model the shared state consists of $smem$ and $caches$ (Figure 3), and in the network model it is the snapshots in the server that is constructed through the network replay function. These represent the global view of the shared state where there may be multiple caches competing to become persistent, but clients observe only a local view, in which just one of the caches is chosen. Because only one client may have ownership of the data at a time (a property enforced by $owner$ in the DSM and by the order that the logical time imposes in the network), it is sufficient to define $\mathbb{R}_{\mathsf{LDSM}}$ such that it holds if the local views of the shared state are equivalent. For the DSM this means $smem$ plus the cache corresponding to $cacheID$. In the network model, the local view of the shared state view is encapsulated in the local state corresponding to the server with the latest snapshot time stamp out of those that acknowledged the client's request.

To test if this definition adequately captures the intended relation, we consider several sample $Log_{\texttt{net}}$s in Figure 7 for which we know what the corresponding DSM should be and check that $\mathbb{R}_{\mathsf{LDSM}}$ is satisfied. The figure illustrates two client nodes and three replica server nodes that are running the Paxos protocol. Note that each client marks the beginning and the end of `prepare` and `write/accept` operations with ghost messages $Ghst$`(·, BEGIN)` and $Ghst$`(·, END)`.

First consider the log **L1**. In this case C1 begins committing update function f at logical time 1, but before it returns, C2 successfully prepares at logical time 2 and begins committing g. Both C1 and C2 then successfully complete their commits. Although C1 returns after C2 in the real-time ordering, the network-based specification forces successful requests to be ordered by logical time. Therefore for C1 we expect the corresponding LDSM to be the result of a strong push of f by C1, and for C2 it should be the same but followed by a successful pull by C2, then another strong push of g. By replaying **L1** we find that

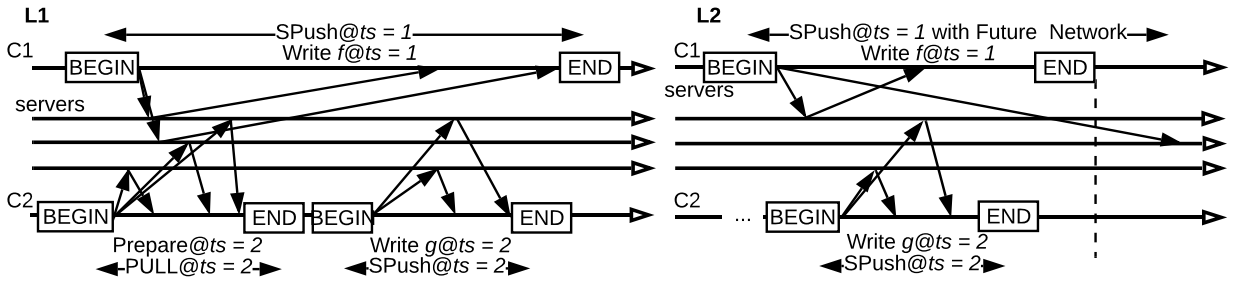

Figure 7: Challenges in Refinement Proofs

C2 must have seen C1's commit before committing g, so the C2's local view of the state will reflect the result of applying f followed by g. Thus **L1** is indeed related to the expected LDSMs by $\mathbb{R}_{\mathsf{LDSM}}$.

L2 is a similar case, but it raises a new problem. Now C1's commit has returned, but at the end of **L2** (marked by the dotted line), the request has only reached one server. C2's commit still clearly maps to a strong push, but because C1 comes first in the logical timeline, we must first assign it an atomic result. However, the request has not reached a quorum of servers so it is not yet persistent and cannot be mapped to a strong push; nor is it definitely rejected so it cannot be treated as a weak push or a failure either. At this point the outcome of C1's commit is truly undecided and it cannot be mapped to any atomic operation. To resolve this issue $\mathbb{R}_{\mathsf{LDSM}}$ needs a method of predicting the network's future to learn an operation's eventual outcome.

Research Task 2a: Designing the Completion of the Network with Oracles Solving the problem presented by **L2** is crucial, because the ADO operations should be mapped to definitive decision in the network-based specification level and this is the key to hide the sophisticated network interleaving in the ADO level. We plan to solve the problem presented by redefining $\mathbb{R}_{\mathsf{LDSM}}$ to hold whenever the completion of a $Log_{\texttt{net}}$'s local view is equivalent to that of a LDSM. We can safely assume that clients of the network-based specification can only update the shared state using `prepare` and `write` operations, and design the operations to rely on a network oracle $\mathbb{O}_{\text{net}}$ to determinize the network. This implies that, for any particular sequence of calls to `prepare` and `write`, the future of the network can be computed. If one assumes an arbitrary function to decide this sequence (a *phase scheduler*, which is a variant of an oracle) then a $Log_{\texttt{net}}$ can be extended arbitrarily to learn the outcome of an operation.

Research Task 2b: Formalizing Refinement Relation and Verifying the Relation The phase scheduler/oracle design and the new definition of $\mathbb{R}_{\mathsf{LDSM}}$ require formal definitions and safety proofs so that they can connect concrete distributed systems to the ADO model. We plan to start from a Paxos-like network specification as in the figure to formalize the refinement relation and verify by refinement proof that our definitions correctly map to the ADO model and extend to other network-based specifications. The core of these proofs rests on showing that the network-based specification satisfies a property that is similar to both the Replicated State Safety of the ADO model (Theorem 1) and the safety properties proved in previous distributed system verification work [9, 20, 31]. We need to prove the following theorem and its refinement relationship to Theorem 1.

Theorem 2 (Network Replicated State Safety)**.** *For all* $Log_{\texttt{net}}$ l *satisfying* $\text{valid}_{\texttt{net}}(l)$*, if global state* $S = \text{Replay}_{\text{net}}$ l*, then for every pair of client snapshots (*$snap$*), one is a prefix of the other.*

Recall that the network-based specification will be parameterized by certain protocol-specific details. Theorem 2 must therefore make some assumptions that the parameters are expected to satisfy. Based on these *protocol invariants*, we will proves Theorem 2 and the lemmas it depends on. We plan to design the

protocol-specific parameters as general as possible such that the safety proof will be a reusable proof template; i.e., once one proves the protocol invariants the rest of the proofs are automatically derived for free.

4 Case Studies (and More on Composition)

Through case studies we aim to demonstrate that ADVERT can verify individual distributed systems as well as a composition of multiple distributed systems. To show the applicability of ADVERT we aim to further verify a novel design of distributed shared memory that is built on top of state-of-the-art network hardware.

4.1 Individual System Verification

To support the claim that the parameterized network-based specification is sufficiently general to model a range of distributed protocols, we plan to instantiate multiple distributed systems and their protocols per network-based specifications.

Research Task 3a: Instantiation of Individual Systems to Improve Network-Based Specifications With our first target network-based specification which models multi-Paxos-like systems, we plan to instantiate Paxos [16], multi-Paxos [28], and Raft [24] as primary systems. The instantiation of network-based specification can be done at the system specification level. One can for example specify a high-level multi-Paxos specification in Coq and map it to the network-based specification by filling out the parameterized components of the network-based specification and verifying that the predefined invariants for both specifications hold.

Studying and instantiating multiple systems are important to validate whether the network-based specification is general enough. The instantiation and revision of the network-based specification can be an interactive task to find the right middle ground. We aim to start from a heuristic method to iterate through the design and find principles and metrics to evaluate whether a network-based specification is properly designed.

Once the network-based specification becomes stable, we aim to instantiate more systems. For example, there are tens of multi-Paxos variants, such as Egalitarian Paxos [22], Vertical Paxos [17], Paxos quorum lease [23], Mencius [21], and so on, which shares similar behavior to multi-Paxos. While verifying individual protocols were considered a great challenge, we aim to verify these variants with ease using our parameterized network-based specification and reusable proofs that connect to the ADO model.

For different classes of network-based specifications we will repeat a similar process to establish a proper network-based specification and verify multiple instances of systems.

Research Task 3b: Code-Level Verification of Individual Systems The goal of ADVERT is not limited to theoretically reasoning about distributed systems, but to verify real systems down to the code level. We plan to connect the low-level code of the system to the high-level specification of the system, which then connects to the network-based specification.

We plan to use the concurrent certified abstraction layer (CCAL) [8] approach, which was developed as part of the PI's prior work. The CCAL approach divides the code into modular layers. The layers are verified individually and then the adjacent layer interactions are verified using contextual refinement proofs. Once all layers and their interactions are verified to be correct, the entire code, from the bottom to the top layers, are guaranteed to be correct. While the CCAL approach modularizes the verification task, it still requires a certain degree of redundant efforts when it comes to verifying different systems and PI's DeepSEA project (funded by NSF CCF, Software & Hardware Foundation) aims to reduce the redundant effort using a language-based approach.

```
ADO Lock:
  DSM[ℤ] owner
  Event acq (id, owner) = if owner
    = ⊥ then id else owner
  Event rel (id, owner)
    = if owner = id then ⊥ else owner
  Method Acquire = this.owner := pull;
    this.owner := push acq
  Method Release = this.owner := pull;
    this.owner := push rel
```

```
ADO Array:
  DSM[{ℤ ↦ ℤ}] arr
  Event ins idx val (_, arr) = arr/[idx := val]
  Method Insert idx val = this.arr := pull;
    this.arr := push (ins idx val)
  Method Append val = this.arr := pull;
    idx := length (replay this.arr);
    this.arr := push (ins idx val);
    return idx
  Method Get idx = this.arr := pull;
    return (replay this.arr)[idx]
```

Figure 8: Key-value store components.

We aim to improve the CCAL approach from a different angle as we verify multiple distributed systems that potentially have similar characteristics. We plan to identify common low-level modules and verification tactics that can improve the automation of the code-level verification.

4.2 Composite System Verification

Once individual distributed systems are verified to refine the ADO model, multiple distributed system interactions can be reasoned purely at the ADO level. The ADO model can facilitate the verification of large-scale distributed systems that build on top of multiple distributed systems. Note that the composite distributed systems of our interest are those that preserve the replicated distributed safety and can be modeled as an ADO from multiple smaller ADOs at a high-level.

Key-Value Store Example We first demonstrate how the ADO model aids in reasoning about composition of distributed systems using a small distributed key-value store example. We present two composite implementations and sketch a proof that both are equivalent to more direct implementation as a single ADO. Note that, for simplicity, we do not consider liveness and assume that clients will continually reissue requests in the event of failure.

The components key-value stores are built on are a distributed lock and array (Figure 8). The lock tracks the ID of the current owner and provides `Acquire` and `Release` methods that are wrappers around `push`, `pull`, and the `acq` and `rel` events. Similarly, the array's methods are primarily wrappers, but `Append` and `Get` also compute the concrete data with the replay function to add functionality beyond the `ins` event. The first key-value implementation, `KVLock` (Figure 9), stores metadata for key-value pairs (e.g., the value's size) and values in separate arrays and uses a lock to synchronize updates. To insert a mapping it acquires the lock, inserts the metadata using the hash of the key as the index, inserts the value at the same index, and then releases the lock. If any operation fails, then the entire method fails and the client must retry.

The second version, `KVNoLock`, is a lock-free implementation. It also stores metadata and values in separate arrays, but instead of storing the metadata and value at the same index, it logs the value and stores a pointer to the value as part of the metadata. During insertion it appends the value to the end of the value array, then stores the value's index in the metadata array.

Finally, `KVAtomic` does not use any other distributed components and is implemented as an entirely self-contained object. Because this is the simplest of the key-value store specifications, we can show the correctness of `KVLock` and `KVNoLock` by proving they are equivalent to it.

First consider `KVLock.Map`. Because it only calls `Insert` while holding a lock, it is trivial to show that this refines `KVAtomic.Map` if `Acquire` does in fact guarantee mutual exclusion. In other words, given any two histories of `Lock` events h and h' such that `replay` $h = id$, and h' does not contain any `rel` events

by id, then `Lock` provides mutual exclusion if `replay` $(h \bullet h') = id$. This can be proved by straightforward induction on the history and observing that only a `rel` event by id can set the owner to $\perp$. Then because the only time when the value and metadata arrays can be out of sync is while the lock is held, there is no danger of this being observed by a concurrent `Map` or `Lookup` call. The proof for `KVLock.Lookup` proceeds similarly because it also always protects the value and metadata arrays with a lock.

In `KVNoLock`, there is no lock to prevent interleaving so atomicity is guaranteed instead by the order in which the arrays are accessed. Suppose that two clients $C1$ and $C2$ call `Map` concurrently with the same key. `Insert` and `Append` are atomic because they are just wrappers around `push` and `pull`. Then the possible ways the `Map` calls can interleave are: 1) `Append1 Append2 Insert1 Insert2`; and 2) `Append1 Append2 Insert2 Insert1` (and the symmetric cases with $C2$'s `Append` coming first). The atomicity of `Append` prevents the two calls from writing to the same index. Thus, no matter in which order the `Insert` calls occur, the key-value mappings will observed in the same order as if the `Map` calls had arrived sequentially.

```
ADO KVLock:
  DSM[{lk: Lock.DSM; vals: Array.DSM;
    meta: Array.DSM}] kv
  Method Map k v =
    this.kv.lk.Acquire;
    this.kv.meta.Insert hash(k)
      sizeof(v);
    this.kv.vals.Insert hash(k) v;
    this.kv.lk.Release
  Method Lookup id k =
    this.kv.lk.Acquire;
    v := this.kv.vals.Get (hash k);
    this.kv.lk.Release;
    return v

ADO KVNoLock:
  DSM[{vals: Array.DSM;
    meta: Array.DSM}] kv
  Method Map k v =
    idx := this.kv.vals.Append v;
    this.kv.meta.Insert hash(k)
      (sizeof(v), idx)
  Method Lookup k =
    (_, idx) := this.kv.meta.Get hash(k);
    return this.kv.vals.Get idx

ADO KVAtomic:
  DSM[{ℤ ↦ (ℤ * ℤ)}] kv
  Event ins k v (_, kv)
    = kv/[hash(k) := (sizeof(v), v)]
  Method Map k v = this.kv := pull;
    this.kv := push (ins k v)
  Method Lookup k =
    this.kv := pull;
    (_, v) := (replay this.kv)[hash(k)];
    return v
```

Figure 9: Key-value store examples.

Research Task 3c: Modeling Different Composition Patterns The example outlines how ADO can simplify the reasoning of distributed systems that consist of multiple distributed components. However, real-world distributed systems can be more sophisticated with respect to how they glue their subcomponents. For example, a system may be partitioned into multiple shards, employ distributed transactions to handle concurrency and request ordering, and use failover and rollback to handle component failures. Luckily ADO can hide the complexity within each subcomponent and allow one to focus on the gluing methodologies.

We aim to extend ADVERT to support the reasoning and verification of different types of ADO compositions. We plan to study and create a reusable specification and proofs that cover commonly used composition patterns. Examples of such patterns include two-phase-commit-like distributed transactions that are commonly found in sharded systems and nested replication that are used in geo-replicated settings.

Research Task 3d: Distributed Shared Memory Verification As a final step to evaluate the ADO model and the entire ADVERT framework, we aim to verify a practical design of distributed shared memory system. This system development is part of co-PI Soulé's research on co-designing network and distributed systems. The goal is to create a petabyte-scale distributed memory system that can be accessed quickly with the help of a programmable switch. The system will be composed of replicated shards which can be modeled as ADOs.

5 Intellectual Merit

Existing distributed system verification efforts have been focused mostly towards handling complex low-level details of individual distributed systems, whereas little attention has been paid towards verifying multiple individual distributed systems and their interactions. The proposed research highlights this overlooked challenges and proposes a novel verification and reasoning approach surrounding an atomic distributed object (ADO). Our research will introduce two new capabilities that have not been explored: 2) the ADO will trivialize the verification of multiple distributed system interactions by hiding the low-level details of the system while exposing enough semantics to reason about the system compositions; and 2) ADVERT will significantly simplify the verification of individual systems with a new approach of capturing common distributed system properties and exposing reusable proofs structures through network-based specifications.

Our research aims to study and verify as many well-established distributed systems as possible to extract the generic properties of the systems and common patterns of their compositions. Our goal is to establish concrete models and verification frameworks for different levels of abstractions and apply the findings to verify conventional and cutting-edge distributed systems down to the code level. Similar to how ADO explains the distributed system behavior at a high-level, each step of our research will cover low to high level details of distributed systems and promote the general understanding of complex distributed systems. We expect the research outcome will greatly contribute to guaranteeing the safety and correctness of large-scale systems and applications that are widely used everyday for example through the cloud.

6 Broader Impact

Our proposed research will contribute greatly to build a safe and bug-free distributed software ecosystem. Especially, the parameterized verification framework and atomic-distributed-object-based composite reasoning of distributed systems capture the core properties of distributed systems. Because most seemingly different distributed systems are developed and optimized while preserving these core properties, our verification approach will be able to host a wide range of distributed systems and tolerate frequent system updates to easily verify the safety properties. In addition, our verification framework will guarantee the safety of large-scale distributed systems that were considered very challenging. We expect that our research will considerably lighten the burden of distributed system testing and debugging by making distributed system verification greatly approachable and applicable.

Furthermore, our research can be easily extended to different domains, such as cyber physical systems and internet of things, where hundreds of devices realize a form of a distributed system. For example, mission critical health monitoring wireless sensor nodes and coordination of self-driving cars and avionics would be able to directly benefit from our research. As computing devices are becoming more and more common, our research will be able to verify the correctness of their interactions and have a greater impact.

We plan to develop innovative distributed system software and formal verification curriculum at both the undergraduate and graduate level. PI Shao recently started to teach a course on language-based security at Yale and the term projects are designed based on initial findings of our proposal. Extending the term project, co-PI Shin is planning to develop a distributed system verification course, where students can gain hands-on experience on both system design and verification. We plan to extend the courses to undergraduate and Master's research program where students can participate in the proposed research. We will make our course materials freely available and encourage our colleagues to use them at other universities.

We are planning to further use our research materials to foster diversity and inclusion. Co-PI Soulé has been specially focusing on supporting Latin American and Hispanic students for few years with scholarship programs, student visit programs, and diversity seminars. The PIs plan to continue these efforts jointly and to recruit underrepresented students to get involved in our research.

Several composite distributed system examples that we plan to model and verify are motivated by experiences in the commercial sector, and therefore has a clear promise for industrial impact. For example, co-PI Soulé has worked closely with Western Digital on a project that uses network hardware to build a new breed of distributed shared memory system. Successful modeling and verification of the system can directly influence the safety and correctness of the production prototype.

The design, implementation, and validation of a software artifact is one of our great contributions to our proposal. We plan to release the software developed in this project under a flexible, open-source license.

7 Contribution to Formal Methods and the Field

Correctly implementing a distributed system has been considered as a great challenge and verifying the system has been considered as a greater challenge not only in the systems but also in the formal verification and software engineering fields. Consequently, the applicability of formal verification has been falling behind the needs which stem from the popular use of large-scale distributed systems and their combinations to host cloud-scale applications. Our proposed research will aid for practical and timely solutions to easily verify the correctness and safety of multiple individual distributed systems and their compositions. ADVERT, which will be equipped with reusable proof structures and an high-level atomic distributed object model, will obviate complex reasoning and become an approachable tool for system developers. The research will advance the theoretical understanding of distributed systems by presenting the details of low to high level logic of standalone and composite distributed systems in a modular fasion.

8 Results from Prior NSF Support

Expedition in Computing: The Science of Deep Specification (PI: Zhong Shao) NSF CCF-1521531, $2,046,445, with Andrew Appel and Lennart Beringer (Princeton), Benjamin Pierce and Stephanie Weirich and Steven Zdancewic (U. Pennsylvania), and Adam Chlipala (MIT), 2015-2020. *Intellectual Merit:* The Yale Component of this expedition project aims to develop a concurrent certified OS kernel (CertiKOS) and connect it with the verified RISC-V hardware (developed at MIT) and the web server (developed at UPenn, and verified at Princeton). The key emphasis is to work out the detailed specification for the machine interfaces (e.g., for RISC-V) and the system call interface (e.g., for CertiKOS) so that software and hardware components verified by multiple DeepSpec groups can indeed be linked togher. During the first two years, The Yale team has successfully developed a clean-slate CertiKOS hypervisor OS kernel that runs successfully on both Intel and AMD multicore platforms with hardware virtualization and can boot Ubuntu or Debian Linux as guest [6]. We have also developed a new compositional approach for formally specifying and verifying sequential and concurrent OS kernels [3, 4, 6]. *Broader Impacts:* This award has partly supported multiple postdocs and students in the past two years. PhD student Ronghui Gu will join Columbia CS as a new tenure track Assistant Professor. The team has organized multiple outreach workshops in 2016-2017, and also a two-week DeepSpec summer school in 2017 with more than 150 attendees from all over the world. PI Shao has incorporated the layered CertiKOS kernel into an innovative undergraduate OS class. *Representative Publications*: two PLDI papers [3, 4] and one OSDI paper [6].

Co-PI: Robert Soulé This is the first NSF proposal for which Soulé is a PI. The funding provided by this grant will help establish his career as a new assistant professor at Yale University.

Co-PI Ji-Yong Shin There is no prior NSF support for which Shin is a PI.

References

[1] Apache hadoop. https://hadoop.apache.org/, 2019.

[2] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[3] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proc. 2016 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 431–447, 2016.

[4] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proc. 2016 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, 2016.

[5] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 328–343, New York, NY, USA, 2017. ACM.

[6] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjoberg, , and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[7] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 653–669, 2016.

[8] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, 2018.

[9] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Apr. 2008.

[11] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[12] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 245–256, June 2011.

[13] T. Kahsai, R. Kersten, P. Rümmer, and M. Schäf. Quantified heap invariants for object-oriented programs. In *LPAR-21, May 7–12, 2017, Maun, Botswana*, pages 368–384, 2017.

[14] J. Kim, V. Sjöberg, R. Gu, and Z. Shao. Safety and liveness of mcs lock—layer by layer. In *Asian Symposium on Programming Languages and Systems*, pages 273–297, 11 2017.

[15] E. Koskinen and M. Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 186–195, New York, NY, USA, 2015. ACM.

[16] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, Dec. 2001.

[17] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, New York, NY, USA, 2009. ACM.

[18] S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin. A survey of open source tools for machine learning with big data in the hadoop ecosystem. *Journal of Big Data*, 2(1):24, 2015.

[19] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL'12*, pages 455–468, 2012.

[20] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 370–384, New York, NY, USA, 2019. ACM.

[21] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[22] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, 2013.

[23] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[24] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[25] I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, Dec. 2017.

[26] J.-Y. Shin, J. Kim, W. Honoré, H. Vanzetto, S. Radhakrishnan, M. Balakrishnan, and Z. Shao. Wormspace: A modular foundation for simple, verifiable distributed systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 299–311, New York, NY, USA, 2019. ACM.

[27] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2014.

[28] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42, 2015.

[29] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[30] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. 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, pages 357–368, New York, NY, USA, 2015. ACM.

[31] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. ACM.