WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems

Ji-Yong Shin
Yale University

Jieung Kim
Yale University

Wolf Honoré
Yale University

Hernán Vanzetto
Yale University

Srihari Radhakrishnan*
Duke University

Mahesh Balakrishnan*
Facebook, Inc.

Zhong Shao
Yale University

ABSTRACT

We propose the Write-Once Register (WOR) as an abstraction for building and verifying distributed systems. A WOR exposes a simple, data-centric API: clients can capture, write, and read it. Applications can use a sequence or a set of WORs to obtain properties such as durability, concurrency control, and failure atomicity. By hiding the logic for distributed coordination underneath a data-centric API, the WOR abstraction enables easy, incremental, and extensible implementation and verification of applications built above it. We present the design, implementation, and verification of a system called WormSpace that provides developers with an address space of WORs, implementing each WOR via a Paxos instance. We describe three applications built over WormSpace: a flexible, efficient Multi-Paxos implementation; a shared log implementation with lower append latency than the state-of-the-art; and a fault-tolerant transaction coordinator that uses an optimal number of round-trips. We show that these applications are simple, easy to verify, and match the performance of unverified monolithic implementations. We use a modular layered verification approach to link the proofs for WormSpace, its applications, and a verified operating system to produce the first verified distributed system stack from the application to the operating system.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Software and its engineering → Formal software verification**; *Distributed systems organizing principles.*

KEYWORDS

distributed registers, distributed system verification, distributed building blocks

*Work done while authors were at Yale University.

ACM Reference Format:
Ji-Yong Shin, Jieung Kim, Wolf Honoré, Hernán Vanzetto, Srihari Radhakrishnan, Mahesh Balakrishnan, and Zhong Shao. 2019. WormSpace: A Modular Foundation for Simple, Verifiable Distributed Systems. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362739

1 INTRODUCTION

Cloud-scale platforms offer developers a number of storage and coordination services that expose simple, data-centric interfaces. At first glance, these services are diverse: they provide different APIs such as key-value stores, block stores, shared logs, object stores, and filesystems. However, the protocols used by these systems to provide properties such as durability, failure atomicity, consistency, and concurrency control are quite similar. Thus, codebases are often highly redundant, re-implementing protocols such as Paxos [33] and Two-Phase Commit (2PC) [22] with slight variations. Each variation leads to different APIs and performance characteristics, but can introduce subtle code and protocol bugs.

In this paper, we explore a data-centric abstraction for distributed systems called the write-once register (WOR). The WOR has a simple API: a client can *capture* a WOR; *write* to a captured WOR; and *read* the WOR. The WOR offers linearizable consistency and is safe for concurrent accesses: if multiple clients attempt to capture and write the same WOR, only one will succeed.

WORs can be naturally implemented via the Paxos protocol (with modifications to support quorum reads), offering durability and availability against a minority of storage servers failing. In fact, the WOR *capture*/*write* API mirrors the phases of single-shot Paxos. WORs can also be implemented via other protocols such as Primary-Backup or Chain Replication [57], obtaining different durability and availability guarantees.

Most distributed services embed WORs, but hide them underneath a higher-level API:

- ***A sequence of WORs*** is often used to impose a total order, but hidden behind restrictive interfaces such as replicated state machines [49, 55], shared logs [3], groups [6, 56], namespaces [9, 30], filesystems, databases [5], or objects [4]. Often, the implementation of the WOR is fused with the machinery that implements the high-level API.
- ***A set of WORs*** represents decisions taken by participants in distributed transaction protocols such as 2PC; the final commit decision for a transaction is a function of these WORs. In

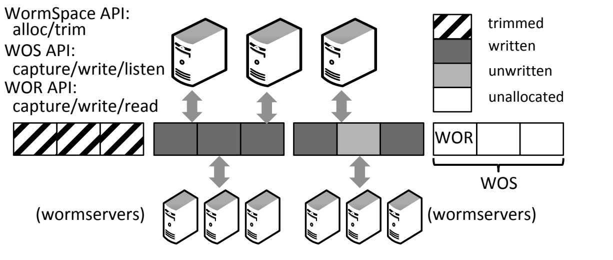

Figure 1: *WormSpace architecture: clients can access a shared address space of write-once registers.*

fault-tolerant protocols, each decision WOR is either layered inefficiently over a replicated state machine, or entwined with a transaction coordination logic [21].

We argue that the WOR should be a first-class system-building abstraction. By providing single-shot consensus via a simple yet versatile data-centric API, the WOR acts as the bottom layer in a modular stack for building strongly consistent distributed systems. The resulting modularity has two benefits. First, it enables *simple* systems: the code and logic for consensus can be provided by a small number of high-quality implementations (e.g., Paxos and Chain Replication) and reused across different systems. Second, it enables *verified* end-to-end systems. With a portable layered verification approach [23, 25], the WOR implementation can be verified once and reused for the verification of applications that use the WOR. The application can be verified easily without dealing with the complexity of distributed asynchrony and failures. Also, the WOR can be layered over a verified OS to enable full-stack verification from the application to the OS.

Accordingly, we present the design, implementation, and verification of WormSpace (contracted from **W**rite-**O**nce-**R**ead-**M**any Address **Space**), which provides applications with a shared address space of durable, highly available, and strongly consistent WORs (see Figure 1). WormSpace divides the address space into contiguous write-once segments (WOSes), which act as coarse-grained units for allocation, notification, reconfiguration, and garbage collection. Internally, each WOR is implemented via a conventional single-shot Paxos instance; WormSpace can be viewed as a system to organize, access, and manipulate these Paxos instances via data-centric APIs. We implement WormSpace via a combination of a client-side library and storage servers. We formally verify the client-side library and the server code written in C using the Coq [14] proof assistant. We verify the functional correctness of the code, as well as distributed properties (e.g., write-once semantics) achieved collaboratively by the client library and the server code.

Applications built over WormSpace consist entirely of capture, write, and read commands on the write-once address space, rather than message-passing protocols. As a result, they are easy to develop and verify. We implement three applications over WormSpace: WormPaxos, a Multi-Paxos implementation; WormLog, a distributed shared log; and WormTX, a distributed, fault tolerant transaction coordinator. All these applications are built entirely over the WOR API, yet provide efficiency comparable to or better than handcrafted implementations. Specifically, we do not 'open the Paxos box' while implementing these applications; the logic for consensus and durability remains strictly contained within the WOR abstraction. In contrast, state-of-the-art implementations for all three applications require the complex melding of Paxos logic with other protocols to obtain efficiency. Further, separating out the WOR enables novel design points: for example, a shared log that uses Paxos (rather than Chain Replication) to replicate each command, supporting appends in just two round-trips in the failure-free case.

WormSpace and its modular WOR design facilitate verification of distributed systems. Contextual refinement, the key technique in a layered verification approach (detailed in Section 2.2) [23], allows for the code above the WormSpace API to be verified easily and incrementally. Applications can be verified without having to deal with the complexity of distributed coordination, which is encapsulated within the WOR layer. To verify an application's correctness against WormSpace, we simply link its proof to the top-most layer proof of WormSpace. Similarly, we can easily link the bottom-most layer proof of WormSpace to CertiKOS [24], a fully verified OS, enabling the first verified system stack from the distributed application to the OS, excluding only the hardware and the network. The linking ensures that verified software components interact with each other correctly as verified without leaving any anomalous corner cases [17]. As a result, we can verify each layer once and reuse the proof multiple times to easily expand the verified code base.

In this paper, we make three contributions. First, we identify the WOR abstraction inherent in many distributed systems and present a simple, data-centric WOR API as a first-class programming abstraction. Second, we implement three distributed applications over this API; for each one, our modular design easily allows new configurations with different performance and availability properties, while matching the performance of an existing monolithic implementation in a similar configuration. Finally, we show that the modular design of the resulting systems, when combined with the layered verification approach, facilitates the reuse of software correctness proofs, and enables verification that crosses distributed system/application boundaries.

2 BACKGROUND

2.1 A Least Common Denominator API

We stated that various systems hide WOR functionality behind high-level APIs. We examine different classes of systems to make two points: most systems are similar in their use of a WOR kernel; but they hide it behind APIs that hinder flexibility, reusability, and performance. While some of these APIs can be implemented over each other, none of them acts naturally as a lowest common denominator for all others. The WOR fills this gap.

State Machine Replication (SMR) / Multi-Paxos [49, 55] systems allow arbitrary (but deterministic) application code to be replicated, via an interface that allows servers to propose new commands and learn them via an upcall. The SMR API is general and easy to use; however, it limits applications by not exposing the underlying address space of WORs. In a sense, SMR imposes a sequential write / sequential read interface on an address space of WORs. The SMR interface can be implemented via multiple protocols; in the other direction, however, Multi-Paxos protocols are exclusively used to support an SMR interface, and designed for the sequential write / sequential read API.

Shared logs [3] provide an append/read API to applications. Unlike in SMR, applications can directly read from WOR instances, examining the history of commands. However, as with SMR, applications cannot directly write to WOR instances; all writes must be funneled through the shared log API, which imposes a total order on commands. In effect, a shared log imposes a sequential write / random read interface on an address space of WORs.

Group communication (GC) systems allow sending messages to groups of servers; each message is atomically delivered with ordering guarantees. Each slot in the total or partial order of messages to the group is effectively a write-once register; the message send primitive acts as a write operation. As with SMR, the GC send/receive API can be viewed as imposing a sequential write / sequential read interface on an address space of WORs.

Coordination services (e.g., Chubby [9] and ZooKeeper [30]) typically expose a filesystem-like API to applications. Such an API is ideal for use cases such as membership management and leadership election, but is awkward for the replication of arbitrary data or general-purpose ordering of commands. These systems are usually implemented over SMR, GC, or shared log APIs.

Transaction coordinators are responsible for coordinating transactions across distributed state. In effect, they are manipulating a set of WORs, each one representing the prepare/abort decision for a participant so that an atomic commit happens across the system. Concurrency control is usually implemented via an orthogonal mechanism such as locking.

We argue that WORs represent a least common denominator interface: all the systems described above can be implemented easily and efficiently over a WOR.

2.2 Verification Approach

Modularity of WOR enables verification based on the certified concurrent abstraction layer (CCAL) approach [23, 25], where we divide the system into modular layers, verify the correctness of each layer independently, and verify the end-to-end behavior of the system via *contextual refinement* between layers. Each layer L is a state machine which has its corresponding implementation i and an execution environment context t. The context t includes programs and configurations that can run on the state machine; and such context is not limited to a sequential program but it can be a concurrent operating system or even an entire distributed system. Informally, a layer L_{low} contextually refines the higher layer L_{high} if each state transition made by L_{high} based on any context t corresponds to a sequence of state transitions by L_{low} which has the context t and L_{high}'s implementation i_{high}. We can formally represent contextual refinement $L_{low} \sqsubseteq_{cr} L_{high}$ as verifying the following:

$$\forall t, L_{low}(i_{high} \oplus t) \sqsubseteq L_{high}(t),$$

where $\sqsubseteq$ is the refinement relation and $\oplus$ computes the union of implementation modules and contexts.

Contextual refinement is powerful since layers can be verified only once independently; and layers can be linked by verifying that each layer refines the layer above it for an arbitrary context. When the stack is extended with a new verified layer on top, the inter-layer contextual refinement proofs can be reused with an updated context to include the new layer. For example, if we add a new layer L_{top} on top of verified layers L_{mid} and L_{btm}, we need one new proof that shows L_{mid} contextually refines L_{top}, but we can reuse the proof for $L_{btm} \sqsubseteq_{cr} L_{mid}$ without requiring any modification to the proof because the proof holds "for all" context t. After the proof of $L_{mid} \sqsubseteq_{cr} L_{top}$, we are automatically guaranteed that L_{btm} contextually refines all the way up to L_{top} as follows:

$$\forall t, L_{btm}(i_{mid} \oplus (i_{top} \oplus t)) \sqsubseteq L_{mid}(i_{top} \oplus t) \sqsubseteq L_{top}(t).$$

Internally, each layer is composed of the C implementation, specifications, and proofs. To develop a layer L_k, the developer writes source code in C; the high-level and the low-level specifications in Coq, which specify how the code changes abstract state and memory, respectively; auto-generates the Coq representation of C source code using CompCertX [23]; and writes three proofs: 1) p_k, a proof that the generated code refines the low-level specification; 2) r_k, a proof that the low-level specification refines the high-level specification; and 3) $R_{k-1,k}$, a proof using p_k and r_k to verify that L_{k-1} contextually refines L_k. The proofs p_k and r_k guarantee that the C code (i.e., its verified Coq representation) is correct as defined by the specifications. With the contextual refinement proof $R_{k-1,k}$, we are assured that the C code in L_k never uses the code in L_{k-1} in an undefined way; calls to C functions in L_{k-1} always return defined results to L_k; and variables used and allocated in each layer have their own memory locations and are safely accessed.

Consequently, proving the contextual refinement relation for each pair of layers in the stack guarantees the functional correctness of the entire system: all layers function correctly from L_{btm} to L_{top} independently and together. With the help of the verified CompCertX compiler, the correctness of system continues to hold even after the C code is compiled into assembly. To build an application on top of a verified system, we simply add layers corresponding to the application on top of L_{top}. The application uses L_{top} as its bottom layer for verification and is oblivious to the layers underneath. The contextual refinement relation between L_{top} and the application guarantees that the application uses the underlying system (from L_{btm} to L_{top}) correctly.

Such *co-verification* of the application and the system is critical, but often overlooked and considered difficult. Without co-verification, the application and the system can be verified independently but still be incorrect as a whole, since the application can abuse the system interface or take actions based on wrong assumptions [17]. For example, for the same write interface, the system and the application may have different address bounds and the application can write beyond the system's address limit. Another example involves slightly different definitions for correctness conditions: a storage system may interpret durability as "flushing to local disk", while the application may expect durability from the storage system to mean "stored on a backup machine"; both can be verified correct, yet the combination will be incorrect. Such mismatches can neutralize the verification effort. Contextual refinement not only guarantees that the application uses the system interface correctly but also guarantees that the application's assumptions about the interface are valid.

```
/* WormSpace APIs */
// allocates a WOS
int alloc(char *metadata, int size, segid_t *newsegid);
//trims a segment
int trim(segid_t seg);

/* WOS APIs */
//batch captures a sub-range within the WOS
int seg_capture(segid_t seg, int *retcodes, off_t start,
                off_t end);
//batch writes a sub-range within the WOS
int seg_write(segid_t seg, char *buf, int size, int *retcodes,
              off_t start, off_t stop);
// registers a listener for write notifications
int seg_listen(segid_t seg, callback_t listener);

/* WOR APIs */
//captures a WOR
int capture(segid_t seg, off_t addr, int *captureID);
//writes a single WOR
int write(segid_t seg, off_t addr, char *buf, int size,
          int captureID);
//reads a single WOR
int read(segid_t seg, off_t addr, char *buf, int size);
```

Figure 2: *The WormSpace API.*

In addition to the functional correctness proof, we verify the distributed protocols and global properties of the entire system that are not immediately visible from the code by adding a *ghost layer*. We add a new network model to the ghost layer and tie together independent nodes in distributed systems to enable the verification of their collective behavior such as distributed nodes maintaining consensus. Although the ghost layer is a logical layer without a C implementation, it is part of our contextual refinement chain where the verified properties are guaranteed to hold in any layer above.

We later show that the verification of WormSpace leads to easy verification of applications on top and can extend the verification of a fully verified OS stack.

3 THE WORMSPACE SYSTEM

The WormSpace API (Figure 1 and 2) provides applications running on client machines with a shared, random-access address space of WORs. All calls in the WormSpace API are safe for concurrent access, providing linearizable semantics for the address space. The address space is divided into write-once segments (WOSes) of fixed size. Segments are explicitly allocated via an *alloc* call that takes in a segment ID and succeeds if it is as yet unallocated. The *alloc* call takes an optional metadata payload to be associated with the new segment. Clients can check a segment to see if it is allocated by some other client, obtaining the metadata if this is the case.

Once a client has allocated a WOS, any client in the system can operate on WORs within the segment. Specifically, it can *capture* a WOR; *write* to it; and *read* from it. Any call to a WOR in an unallocated segment fails with an error code. Clients must capture an address before writing to it to coordinate replicated servers to make the write atomic and immutable. The capture call is similar to a preemtable lock (e.g. prepare of Paxos): the lock must be acquired to write, but it can be stolen by others.

A successful capture call returns a unique, non-zero captureID; a subsequent write by the same thread is automatically parameterized with this ID, and succeeds if the WOR has not been captured by some other client in the meantime. Alternatively, threads, processes, and even clients can capture a WOR and then hand over the captureID to some other thread/process/client that passes it in explicitly as a parameter to a write, allowing the capture and write to be decoupled in space. Finally, a write parameterized with a captureID of 0 does not require a prior capture; we call this an *unsafe write*. Unsafe writes are fast because capturing is unnecessary, but not safe for concurrent access; applications must ensure that at most one client issues an unsafe write to a particular WOR.

The WOS provides *seg_capture* and *seg_write* APIs, which act as batched operations, capturing all the WORs in the segment or writing a single value to all of them. A client can also receive notifications when WORs in a particular WOS are written to, via the *seg_listen* call. Garbage collection can be triggered by the application via the *trim* call, which trims individual WOSes. WormSpace returns an error code when a trimmed address is accessed.

3.1 Design and Implementation

WormSpace is implemented via a combination of a client-side library exposing the API shown in Figure 2 and a collection of servers, which we call wormservers (Figure 1). In a sense, the WormSpace design is similar to a distributed key-value store: WORs are associated with 64-bit IDs (consisting of segment IDs concatenated with offsets within the segment) and mapped to partitions, which in turn consist of replica sets of wormservers. Partitioning occurs at WOS granularity; to perform an operation on a WOR within a WOS, the client determines the partition storing the segment (via a modulo function) and issues the operation to the replica set.

Each WOR is implemented via a single-shot Paxos consensus protocol, with the wormservers within a partition acting as a set of acceptors. In the context of a single WOR, the wormservers act identically to Paxos acceptors [34]; a *capture* call translates to a phase 1a prepare message, whereas a *write* call is a phase 2a accept message. The *read* protocol mirrors a phase 1a message, but if it encounters a half-written quorum, it completes the write. Each wormserver maintains a map from WOR IDs to the acceptor state for that single-shot Paxos instance. If a map entry is not found, the WOR is treated as unwritten.

Above this basic WOR interface, the client-side library layers the logic for enforcing write-once segments. Each WOS segment is implemented via a set of data WORs, a single metadata WOR, and a single trim WOR. Allocating the WOS requires writing to the metadata WOR. If two clients race to allocate a WOS, the first one to capture and write the WOR wins.

The *trim* call for garbage collection is implemented via a special message where the client instructs the wormserver to return errors on requests for affected WORs, and delete all states of the WORs. The trim WOR in each WOS enables consensus on a trim command. On subsequent reads or writes to a trimmed WOR, if a subset of the accessed quorum replies that the ID is trimmed, the client-side library completes the trim by issuing it to the remainder of the quorum, and then returns an *E_TRIMMED* error to the application.

Reconfiguration Replacing a minority of wormservers from a partition requires a reconfiguration protocol along the lines of Vertical Paxos [35]. In essence, a reconfiguring client 'seals' the existing configuration by contacting a majority of the servers. The servers promise to respond with errors to messages sent by clients with the existing configuration to prevent progress using this configuration. A new configuration is installed at an auxiliary location; this could be an external membership service, a different partition of the WormSpace deployment, or a different instance of WormSpace altogether. Clients that receive error messages from servers due to a sealed configuration must go check this location for the new configuration, and reissue the command to the new set of servers in the partition.

Alternative WOR implementations Within each WormSpace partition, wormservers can be organized in different ways to realize other consensus protocols. For example, instead of Paxos, we access the wormservers via a client-driven variant of Chain Replication used in CORFU [3]. The client captures and writes to each server in the chain in sequence, and issues reads to the tail. Such a protocol has the benefit of efficient reads which contact a single server rather than a majority quorum, and provides durability against f failures with $f + 1$ nodes rather than $2f + 1$. The downside is the increased write latency, which is linear in the number of servers, and unavailability for writes if a single server goes down until a reconfiguration. In our implementation, we did not implement the CRAQ [54] optimization, which allows for reads to go to any replica instead of the tail. We call our two implementations chain-WOR and paxos-WOR.

WORs could be implemented via Byzantine consensus [10, 12]; we leave this for future work. Note that consensus cannot be realized using a single round trip under asynchronous networks [29]. The capture API is intended to encapsulate such extra coordination for different implementations.

In a sense, the WOR is analogous to the logical block device abstraction found at the bottom of a single-machine storage stack. The WOR simplifies the construction of systems such as shared logs and SMR/Multi-Paxos by hiding the complexity of asynchrony and failures; a block device simplifies the construction of filesystems by hiding the complexity of storage hardware. Following this analogy, it is possible to implement the WOR itself over a shared log or SMR/Multi-Paxos (similar to how a block device can be implemented over a filesystem). However, the more conventional layering places the WOR at the bottom and this simplifies the higher-level system design (Section 4) and verification (Section 5).

4 WORMSPACE APPLICATIONS

To illustrate how WormSpace simplifies applications, we present WormPaxos, WormLog, and WormTX.

4.1 WormPaxos over WormSpace

In principle, implementing Multi-Paxos over WormSpace is simple: the sequence of commands is stored on the WormSpace address space. WormPaxos is an implementation of Multi-Paxos over WormSpace, exposing a conventional state machine replication (SMR) API to applications. In WormPaxos, servers that wish to replicate state act as WormSpace clients; we call these WP-servers. They can *propose* new commands by preparing and writing to the next free address; and *learn* commands by reading the address space in sequential order. If a client finds that the current tail is at the end of a WOS, the client allocates a new WOS and then writes to the next address.

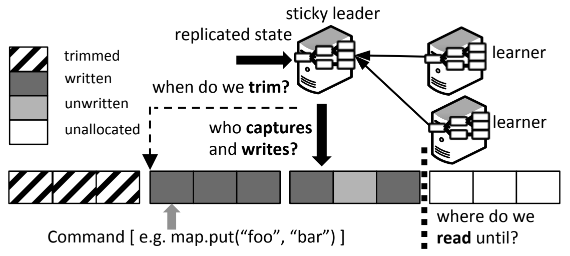

Figure 3: *WormPaxos: servers replicate state by ordering proposals on the WormSpace address space.*

The chief benefit of this layered design is extreme simplicity; the Multi-Paxos consists of a few hundreds of lines of code, which calls data-centric commands over the WormSpace address space. This design also enables flexibility along a number of dimensions (Figure 3):

Flexible Consensus *(i.e., how is the WOR implemented?)*: Consensus in WormPaxos is hidden under the WOR abstraction and can be implemented via many different protocols, ranging from variants of Paxos, atomic broadcast protocols such as Zab [31], and protocols such as Primary-Backup and Chain Replication. In contrast, existing Multi-Paxos designs weld together the single-decision consensus engine – typically Paxos – with the state machine replication machinery responsible for consistency and availability. For example, the WormPaxos codebase can run with no modification over a different WOR implementation such as the chain-WOR; in contrast, existing Multi-Paxos implementations require extensive modification to run over a different single-shot consensus protocol.

Flexible Leadership *(i.e., who calls capture?)*: Sticky leadership – i.e., retaining a single leader across multiple commands – is a key performance imperative for Multi-Paxos implementations, since it allows commands to be decided within a single round-trip rather than two in the absence of failures, and eliminates contention between leaders. In many Multi-Paxos implementations, leadership strategy is baked into the system design; for example, Raft [46] is explicitly designed to support sticky leadership as a first-class consideration. In WormPaxos, a WP-server becomes a sticky leader simply by using a batch *capture* on a WOS; accordingly, leadership strategies such as sticky leader, rotating leader, etc. can be implemented simply as policies on who should call the batch *capture* and when. Further, the leader's identity can be stored within the metadata for each segment, obviating the need for WormSpace to know about the notion of a leader or the leadership strategies involved. If the leader crashes, a new leader that allocates the next WOS can batch *capture* the WOS of the previous leader, complete partially finished operations, and fill in junk values [3] to unwritten WORs to prevent holes in the SMR/Multi-Paxos log.

Flexible Durability *(i.e., when is trim called?)*: By varying when it calls *trim*, WormPaxos can employ different strategies for durability. For instance, a WP-server can *trim* a prefix of the WormSpace as soon as a certain number of WP-servers have seen it, or some WP-server has stored a snapshot in an external data store; this

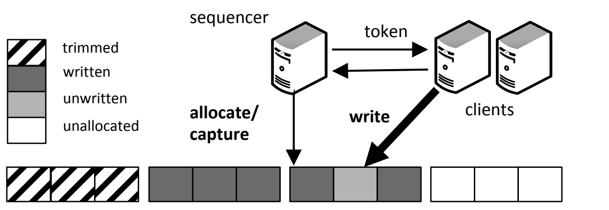

Figure 4: *WormLog: clients can append by obtaining a token from the sequencer and writing to WormSpace.*

information can be piggybacked on new commands appended to the address space. In contrast, existing Multi-Paxos designs are tied to a particular strategy for durability (e.g., when all replicas have seen a command [55]).

Flexible Consistency *(i.e., what addresses do we write and read?)*: WormPaxos derives consistency properties such as linearizability, sequential consistency, or eventual consistency via strategies for writing/reading to the address space. The state at each WP-server reflects some subset of updates in the WormSpace. For linearizable writes and reads, each command has to locate a slot after any completed writes in the address space, but before any empty slots that could be filled by later commands. For a weaker guarantee such as sequential consistency, WP-servers can allocate separate segments and write to them in parallel [40]. Similarly, causal consistency can be obtained by ensuring that new writes from a WP-server go to a later address than any it has already seen. For these weaker consistency guarantees, the random write / random read nature of the WormSpace API allows us to parallelize proposing in a way that we could not do over a conventional SMR (sequential write / sequential read) or shared log (sequential write / random read) interface.

4.2 WormLog over WormSpace

A shared log is a shared address space that provides an *append* / *read* API to clients. CORFU [3] is a system that implements a shared log API over a set of write-once addresses. To append a new entry to the shared log, a client first contacts a centralized sequencer machine to reserve and increment a tail position on the address space. It then issues a write to a write-once address. In CORFU, each write-once address is implemented via a client-driven variant of Chain Replication, where the client writes to each replica in sequence. The write-once semantics are derived by using the head replica of the chain to arbitrate between competing writes to the same address. A key aspect of this design is that the sequencer is merely a soft-state hint about the tail of the log, and does not have to be durable or available.

Achieving a CORFU-like design over WormSpace is straightforward: we simply have each client contact a sequencer node when it wants to append an entry, obtain a slot in the WormSpace address space, and then write to that position (Figure 4). With this design (which we call WormLog), we obtain the two properties that differentiate a shared log from a Multi-Paxos system [42]: the decoupling of sequencing from I/O, since the sequencer does not see the append payload; and the time-slicing of individual commands over different replica sets, assuming that the WOS size is small compared to the volume of in-flight appends in the system.

WormLog addresses a problem with the CORFU system's use of Chain Replication: appends no longer take latency linear in the number of replicas, since they simply issue a WormSpace capture/write, which in turn invokes the Paxos two-phase protocol. However, the WormLog design described thus far takes three round-trips: one to the sequencer, one to capture the WOR, and one to write to it. By decoupling I/O from sequencing, we lose 'sticky leadership'; we can no longer perform a batch capture on the WOS and write to the WOR in a single round-trip, since multiple clients are writing to a single WOS.

Eliminating this extra round-trip is simple. The sequencer allocates WOSes before handing out sequence numbers to clients. The sequencer also pre-captures the WOS and provides the client with the captureID as a token; the client can then predicate its write with this captureID. Accordingly, WormLog realizes a CORFU-like design that uses Paxos (reducing latency to 2 round-trips from the $N + 1$ required by client-driven Chain Replication).

4.3 WormTX over WormSpace

Two-Phase Commit (2PC) [52] solves the transaction commit problem via a transaction manager (TM). Any participant (RMs, or resource managers) that wishes to initiate a commit contacts the TM (message delay #1). The TM contacts all participants to elicit a yes/no vote (#2). Each RM votes, records its vote in local stable storage and responds to the TM (#3). The TM makes a decision based on the votes it receives, and sends back a commit or abort command to the RMs (#4). The TM's decision can be a deterministic function of the RM votes – i.e., the decision is yes if all the votes are yes. Alternatively, the TM can decide no even if all the votes are yes, in which case it stores its decision in stable storage before sending the decision.

The failure model for 2PC is that nodes – TMs or RMs – can crash, but will subsequently come back online. 2PC is known to be a blocking protocol in the presence of such failures. In the case where the decision is deterministic, if a single RM fails – after it has locally stored its vote in stable storage, but before it has responded to the TM – then the protocol has to block until the RM comes back online. In the case where the TM fails – after storing its final decision in stable storage but before sending commit messages – the protocol has to block until the TM comes back online. In both cases, the remaining RMs cannot determine the decision.

We consider making the deterministic (i.e., the TM does not have a separate vote) version of 2PC non-blocking. We come up with a number of variants that use WORs. We describe them below and in Figure 5.

[Variant A8: 8 message delays] An obvious solution is to simply store the votes in a set of per-RM WORs. If the TM decision is non-deterministic, a WOR is used to store the decision as well. In the WOR-based 2PC protocol, an RM initiates the protocol by contacting the TM (message delay #1); the TM contacts the RMs (#2); they capture the WOR (#3 and #4), and then write to it (#5 and #6); send back their decision to the TM (#7), which sends back a commit message to all the RMs (#8). This corresponds exactly

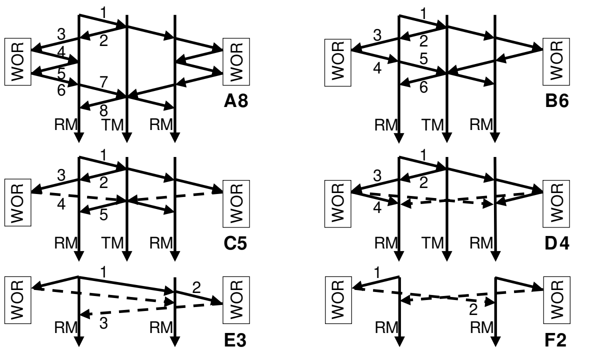

Figure 5: *WormTX: WOR-based non-blocking atomic commit protocols. Dashed arrows are notifications.*

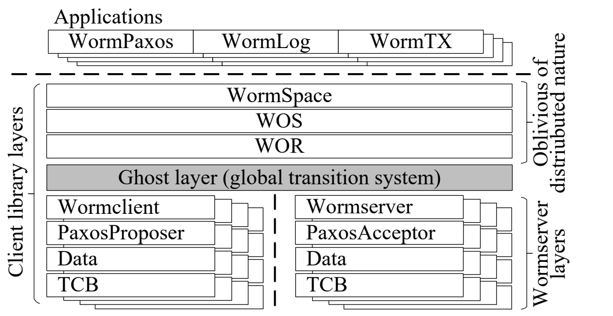

Figure 6: *Layer diagram: client and server stacks are combined as a distributed system in the ghost layer and the distributed nature is invisible from the WOR layer.*

to using Paxos as a black box to replicate the RM decisions for availability; we condered this design as our baseline.

[Variant B6: 6 message delays] A simple optimization involves eliminating the capture messages from the critical path. Each RM can allocate a dedicated WOS for its decisions and batch capture the WOS in advance. This eliminates delays #3 and #4 from variant A8, bringing us down to 6 message delays.

[Variant C5: 5 message delays] Further, rather than have the RM wait for an ACK on the write (message delay #6 in variant A8) and relay it to the TM (#7 in A8), the TM can directly observe the decision by listening for write notifications on the WOS. This compresses #6 and #7 of variant A8 into a single step, bringing us down to 5 message delays.

[Variant D4: 4 message delays] Finally, rather than have the TM wait to be notified of all the WOR writes and then send out a commit message to all the participants (#8 of variant A8), individual RMs can directly listen to each other's WOSes; this brings us down to 4 message delays.

This progression of increasingly fast protocols exactly matches the description by Gray and Lamport [21]; they too proceed from an unoptimized 8-step protocol to an optimized 4-step one in identical fashion, via 6-step and 5-step protocols. In their case, this is achieved by opening up the Paxos protocol and rewiring the flow of requests and ACKs between the various Paxos roles of acceptors, leaders, proposers, and learners. In our case, the optimizations are achieved via the WormSpace API, without requiring any knowledge of the Paxos protocol.

[Variant E3: 3 message delays] We now observe that we do not need a TM, since the final decision is a deterministic function of the WORs, and any RM can time-out on the commit protocol and write a no vote to a blocking RM's WOR to abort the transaction. The initiating RM can simply contact the other RMs on its own to start the protocol (combining #1 and #2 of variant A8), bringing down the number of delays to 3. Interestingly, this variant is not described by Gray and Lamport.

[Variant F2: 2 message delays] Finally, if RMs can 'spontaneously' start and vote, we eliminate delays #1 and #2 of variant A8, bringing the protocol down to two delays, the theoretical minimum for atomic commit. Since this is not a realistic assumption for many systems, we choose variant E3 as our final solution.

Our protocol is in contrast to other non-blocking commit protocols, which require complex message passing logic [52]. Instead, we assemble a non-blocking protocol via simple, data-centric commands on WORs.

Concurrency Control: So far we have described a non-blocking atomic commit protocol built using WORs. To implement distributed transactions with transactional isolation and atomicity, this protocol requires some form of concurrency control. We do not discuss concurrency control schemes in detail because they vary depending on user needs and do not affect the message delays for atomic commit. For completeness, however, we implemented a simple concurrency control protocol based on locking that uses a write-ahead log and Immediate-Restart [1] for deadlock prevention.

Consider variant E3. The server that performs a transaction notifies all servers involved. Each server tries to acquire a lock on its local data for the transaction. If it succeeds, the server writes a write-ahead log and then a yes vote to its WOR. Upon failure to lock, the server aborts the transaction by writing a no vote to its WOR. If each server receives yes ACKs for its own yes write from all servers involved, it updates the data and releases the lock. Otherwise, it releases the lock without the update. This protocol provides strict serializability and failure atomicity.

5 FORMAL VERIFICATION

WormSpace acts as a foundation for verifying distributed systems. We verify WormSpace once and reuse its proof for verifying systems built on top while hiding the complexity of distributed protocol verification. To do so, we extend the Certified Concurrent Abstraction Layer (CCAL) approach [23, 25] introduced in Section 2.2, modeling an asynchronous network of distributed nodes. We apply CCAL beyond a single system verification for the first time and link the proof of WormSpace, applications and a verified OS.

5.1 Layer Structure for Verification

WormSpace consists of two separate stacks of verification layers, the client library (17 layers) and the wormserver (2 layers), over a common set of base functionalities (5 layers). While the number of layers may seem excessive, it matches a conventional software stack designed for modularity: each layer is a C component implementing some interface. A simplified layer diagram is shown in Figure 6.

Both stacks share a common set of base layers: the bottom layer provides an interface to the trusted computing base (TCB), including network communication functions and a small number of system calls. Above this bottom layer, we introduce a data layer which implements various data structures over the trusted primitives. Above the data layer, the client and server stacks diverge. The server stack includes Paxos acceptor layers and the wormserver code above it. The client stack includes layers for Paxos proposer logic and a wormclient layer that issues individual Paxos proposals.

The ghost layer composes these two stacks into a single transition system that models and enables reasoning about multiple wormservers and clients. The ghost layer includes a global state transition system that can reason about all concurrent client and server interactions based on a network model. Safety properties of Paxos (i.e., the write-once guarantee of WOR) are proved in this layer. The contextual refinement proof between the ghost layer and the composition of wormserver and wormclient provides a powerful guarantee for the layers built on top of the ghost layer. Any layer that the ghost layer contextually refines is guaranteed to be correct with respect to both client and server layers. It is guaranteed that any concurrent behaviors of distributed nodes using the client and server layers are correct. Verified distributed protocol properties hold in higher layers while complex proofs are encapsulated in the ghost layer.

Verification above the ghost layer is as easy as verifying a sequential program. For example, the top-level specification for a write in WormSpace is simply translating the global address to a segment address and offset and passing the captureID (cid) to call the lower-level *write* which is already proved safe under concurrent distributed accesses:

```
Func WormSpace_write (addr: Z) (val : Payload)
 (cid : Z) (adt : EnvVars) : option (EnvVars * Z) :=
  let segment:= addr / WOS_SIZE in
  let offset := addr mod WOS_SIZE in
  write segment offset val cid adt .
```

We verify the WOR, the WOS and the WormSpace abstractions. The client stack can be extended to applications such as WormPaxos, WormLog, and WormTX.

5.2 Network Model

To model a real-world network and to prove distributed properties about the system, we employ techniques from concurrency verification [25, 32]. Our network model uses the rely-guarantee reasoning [38] and includes two basic primitives, *send_msg* and *recv_msg*, which manipulate the modeled network state. The model includes a logically linearized sequence of network operations, which we call the global network log. Each distributed node can extract its local interaction with the network from the log, and the log is used to reason about the interaction between nodes.

However, we depart from single-node concurrency verification by modeling the network as unreliable (but non-Byzantine). In our model, *send_msg* simply creates a SEND event in the log, while *recv_msg* creates either TIMEOUT (this models dropped packet) or RECV events in an arbitrary future location (this models packet delays) than the SEND event in the log. In between a pair of SEND and TIMEOUT/RECV, any other nodes can freely record their operations (this models packet reordering). A RECV after a SEND does not necessarily mean that the RECV event received the value sent by this SEND. The actual value can be a duplicate message from a previous send (this models duplicate packets).

```
Function WOR_ghost_write (addr: Z) (val : Payload) (cid : Z)
 (adt : EnvVars) : option (EnvVars * Z) :=
  let net_l := adt . net_l (* get net log from Env context *)
  let nid := get_node_id adt in (* get current node id *)
  (* replay the net log; get the local node state ; and
     check if the node is in a writable status *)
  if (can_write ((replay_log(net_l))[nid]) addr val cid) then
    (* log write intent with a ghost msg to the net log *)
    let net_l1 := (ghost_write nid addr val cid) :: net_l in
    (* broadcast msgs and collect acks: reflect behaviors
       of other nodes to add send/recv events by this and
       other nodes to the net log *)
    let net_l2 := bcast_n_recv nid addr val cid net_l1 adt in
    (* replay the net log to compute global state ; get
       node's local state ; and check the quorum status *)
    let result := is_qrm ((replay_log(net_l2))[nid]) addr in
    (* log the result using a ghost msg to the net log *)
    let net_l3 := ( ghost_result nid result ) :: net_l2 in
    (* return the updated net log and the result *)
    (adt{net_l := net_l3}, result )
  else None.
```

Figure 7: *A simplified log construction function. It logs local and network events of a node to the network log and calls the log replay function to check state changes.*

5.3 Proving Global Properties

The global state transition system in the ghost layer models a distributed system with multiple concurrent Paxos clients and acceptors from the viewpoint of the global network to enable the distributed protocol verification. It includes (network) log construction functions, a (network) log replay function, and a global state. The log construction function models how each client/server operation affects the network; it governs the communication pattern of each node in the network log to define the Paxos protocol. The log replay function constructs the global state – which is a snapshot of the entire distributed system state or a combination of Paxos-related states in all nodes – by interpreting network events in the network log. Log construction and replay functions are derived from wormclient and wormserver specifications and their refinement relations for the derivation are verified.

Log construction functions interact with the network log and the global state to introduce new network events in the network log. To record local state changes of a node which do not involve network operations, ghost messages are written to the network log. Log construction functions use the log replay function to learn and use state changes incurred by other concurrent nodes and itself. Figure 7 shows an example of how a log construction function which corresponds to a *write* interacts with the network log from the global environmental context (EnvVars) and creates new network events.

The log replay function by itself can replay all behaviors and state changes of a distributed system step by step from the global network log. Based on this capability we prove the Paxos-based safety property of WormSpace:

Theorem 1. *Once a value is written to a WOR, the value in the WOR never changes.*

To prove Theorem 1, we prove the key lemma:

Lemma 1. *Given a valid network log ℓ, if there exists a Paxos round n where a value v is successfully written to a WOR r, any following write to r in Paxos rounds $n' > n$ in the log ℓ can only attempt to write $v' = v$.*

The valid network log is the log that preserves verified invariants such as communication patterns derived from log construction functions. Lemma 1 is proved by induction on writes in the log using other supporting lemmas: e.g., n' is unique and is monotonically increasing, the Paxos-phase-1a/capture at round n' on r returns the written value v, etc. Based on Theorem 1, the immutability and uniqueness of WOS allocation (including leader/sequencer election of WormPaxos/Log) and trim are easily verified.

5.4 Top-Level Theorem of WormSpace

The top-level theorem that we prove for WormSpace is,

Theorem 2. $\forall t, L_{TCB}(i_{AllWS} \oplus t) \sqsubseteq L_{WormSpace}(t)$,

where t is the context and i_{AllWS} is the implementation of all WormSpace layers combined. The contextual refinement proof between all adjacent layers are used as lemmas to guarantee the correctness of the entire code. Theorem 2 also guarantees that the verified Paxos properties in the ghost layer (e.g., Theorem 1) hold for the WormSpace implementation.

5.5 Reusability and Linking

Because the ghost layer encapsulates the distributed nature of WormSpace, the verification of WormPaxos, WormLog, and WormTX does not have to reason about complex Paxos proofs. The verification of any additional distributed protocols above WormSpace reuses the same network model, but requires a new ghost layer. Protocols at different levels of the stack are independently verified within separate ghost layers; invariants of interfaces to the protocol and contextual refinement proofs guarantee non-interference among protocols.

The top-level theorems that we prove for WormPaxos, WormLog, and WormTX are in the same format:

Theorem 3. $\forall t, L_{WormSpace}(i_{App} \oplus t) \sqsubseteq L_{App}(t)$,

where App can be one of WormPaxos, WormLog, and WormTX. By reusing Theorem 2 and transitively combining it with Theorems 3, applications are guaranteed to be correct with respect to all layers of WormSpace and to encapsulate verified Paxos properties. Similarly, Theorem 2 can be reused to verify any system in Section 2.1 and beyond to guarantee WOR semantics, if we use WormSpace as a building block.

WormSpace	WormPaxos	WormLog	WormTX (C5)
4,551	359	362	547

Table 1: *C lines of code for WormSpace and applications. C5 has the largest size among WormTX variants.*

To enable end-to-end verification of WormSpace, WormPaxos, WormLog, and WormTX, we link WormSpace to CertiKOS. The linking requires contextual refinement proof between two interfacing layers. When linking independently developed and verified software pieces together, it is important to check that the specification exposed by the lower layer matches the expectations of the higher layer. Since WormSpace and its applications were codesigned, such a consistency check was unnecessary, but linking WormSpace to CertiKOS required careful consistency checks. Once we link WormSpace with CertiKOS the correctness of WormSpace and the applications is guaranteed from the bottom-level (L_{x86asm}) of the OS without any side-effects [17]; this verifies and guarantees,

Theorem 4. $\forall t, L_{x86asm}(i_{CertiKOS} \oplus i_{AllWS} \oplus i_{App} \oplus t) \sqsubseteq L_{App}(t)$.

The extensibility of WormSpace verification to applications and the OS is difficult for other verified systems [27, 45] to achieve. Especially, it is unnatural and difficult to support contextual refinement, which is based on high-order logic, when the verification tool is based on a SMT solver or first-order logic [13, 36].

5.6 Discussion

The verification of WormSpace relies on a trusted computing base (TCB) consisting of the operating system (OS), the hardware, and the network. However, when we link our verification to CertiKOS, the TCB consists of only the hardware and the network. Our verification tool chain guarantees the correctness up to the assembly level, in contrast to other work that often generate high-level code that require untrusted transformation to low-level executable code [45, 50, 59]. A small part of our system remains unverified: the reconfiguration in WormSpace and the Chain Replication WOR.

6 EXPERIENCE

The main benefit of WormSpace is that compared to Paxos, developers do not need to reason about or understand Paxos protocols to build applications on top, and compared to other fault-tolerant replicated systems, the developer has the flexibility to choose low-level implementation details.

WormSpace applications are easy to build, relying largely on simple invocations on the data-centric WormSpace API to store data durably and to coordinate across machines. The effort taken to implement WormTX was similar to implementing a non-fault-tolerant version. In other cases, WormSpace simplified application-level coordination. The leader election scheme of WormPaxos and the failure recovery scheme for WormLog sequencer are implemented with the WOS *alloc* call: it ensures that among multiple concurrent nodes that try to become the new leader or the sequencer, only one succeeds. The C lines of code to build WormSpace and the applications are summarized in Table 1.

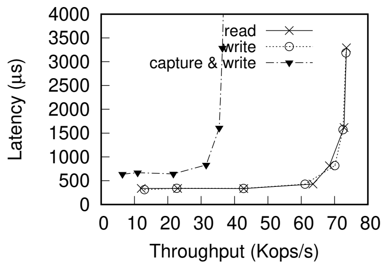

Figure 8: *Microbenchmarks: read/write saturates a single wormserver.*

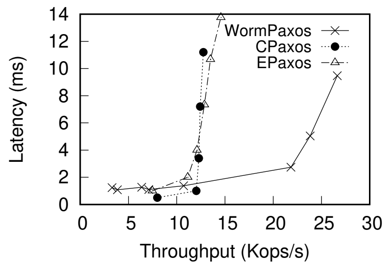

Figure 9: *A verified C-based WormPaxos outperforms unverified Go-based E/CPaxos.*

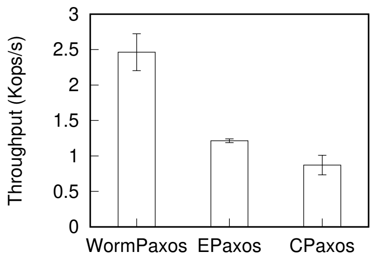

Figure 10: *Fewer writes to persistent storage per operation makes WormPaxos outperform E/CPaxos in persistent mode.*

Although functionally-equivalent applications could be built on top of existing fault-tolerant replicated systems (e.g., an SMR), WormSpace provides clear advantages over such systems. The WormSpace API that maps to a lower-level system behavior naturally allows for a more optimized design (e.g. as demonstrated for WormTX). To reach the level of optimization provided by the WormSpace API, most existing systems require modifying the system itself (e.g., to rewire the communication pattern); this would necessitate a greater effort. Additionally, verifying such existing systems as a whole and extending the correctness proof to the application would remain as great challenges [27, 59], whereas WormSpace is designed to solve both problems.

Our experience with verification was similar to application development, where the verification of WormSpace facilitates that of applications. Our Coq-based verification cannot be fully automated, but the CCAL framework provides templates and libraries that dramatically reduce the proof effort. The entire Coq verification code size is 108K lines. Overall, it took 6 person months to verify WormSpace: 4.5 person months to prove functional correctness and 1.5 person months to prove properties in the ghost layer. Yet, verifying WormPaxos, WormLog, and WormTX, linking these applications to WormSpace, and linking WormSpace to CertiKOS took in a total of 5 person weeks. The proof effort for WormSpace was not small, but reusing the proof for the application was easy. We believe the verification stack can be extended easily (e.g., a key-value store layer on top of the WormPaxos layer), the same way that WormPaxos, WormLog, and WormTX were verified over WormSpace and CertiKOS.

7 EVALUATION

We evaluate the performance of WormSpace and show that verified systems can be as fast as existing unverified systems. Our evaluations demonstrate the potential for using verified distributed code in real life without any slowdown. We run the experiment in two modes: the verified WormSpace stack over a commodity unverified OS (on Amazon EC2, on m4.xlarge instances running Ubuntu 14.04), unless mentioned otherwise; and an end-to-end verified stack running over CertiKOS on a local cluster. We run three wormservers and up to sixteen client nodes. WormSpace has in-memory and persistent modes, which determine whether the data is stored in memory or in persistent storage; in-memory mode is used by default. The data size we use for all experiments is 8 bytes. We focus on the write-related workloads as reads can be massively parallelized – for example, by using a proxy node – in all applications that we use.

7.1 Micro-benchmarks

We use a micro-benchmark to test the base performance of WormSpace (Figure 8). We evaluate the performance of reads and writes. We first pre-fill the address space with data and have clients read different parts of it sequentially. We increase the number of concurrent clients to get different throughput/latency points. A read to a WOR entails 1 RTT between the client and wormservers. The read latency stays low at around 250 microseconds when the load is low and the throughput saturates at about 70K/s operations, which is the peak capacity of a single wormserver.

Similar to the read experiment, we have clients write to a disjoint set of WORs so that clients do not contend to write on the same WOR. We measure two different cases where each client issues a capture to individual WORs before a write, and another case where clients are writing to WORs that are already captured in a batch. The latter is equivalent to writing to a WOS that is captured or doing an unsafe write. The overhead of incorporating a capture call, which adds 1 RTT on every write, doubles the latency and halves throughput compared to issuing writes on batch-captured WORs.

7.2 WormPaxos

To evaluate the verified WormSpace application performance, we compare WormPaxos against the unverified open source code of the Egalitarian Paxos (EPaxos) paper [43]. Under the same configuration, Figure 9 compares the write performance of WormPaxos against EPaxos and the classical Paxos (CPaxos) that is used in the EPaxos evaluation. CPaxos shows slightly lower latency than WormPaxos but the maximum throughput of WormPaxos is much higher than the others. The performance difference comes from different implementations, WormPaxos in C versus the others in Go, and an extra commit phase that exists in E/CPaxos. E/CPaxos asynchronously notifies all acceptors about the written value after the two Paxos rounds, whereas WormPaxos omits this step because WormSpace clients use a quorum read. We also measure the throughput with data persistence on an Amazon EBS GP2 SSD (Figure 10). The absence of the commit phase, makes WormPaxos

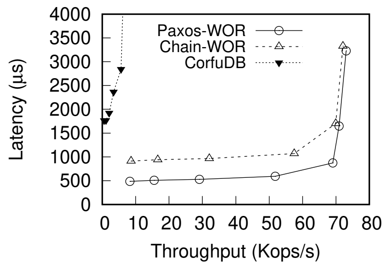

Figure 11: *WormLog: Paxos-WOR can optimize the latency of a shared log design.*

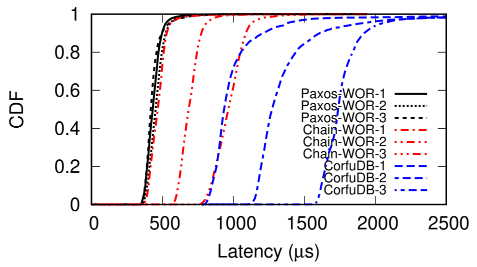

Figure 12: *WormLog latency distribution: Paxos-WOR has constant 2 RTT latency regardless of the number of wormservers.*

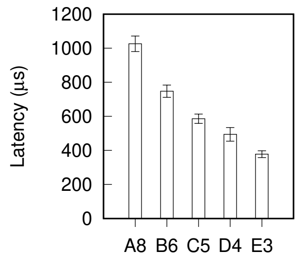

Figure 13: *WormTX: optimizations above WormSpace are easy and enable lower latency.*

achieve higher throughput. Our point here is not to claim WormPaxos runs faster than EPaxos, which internally does dependency checks and ordering, but to show that verified code is not necessarily slow and can run on par with unverified code depending on the implementation choice.

7.3 WormLog

We evaluate the performance of WormLog with Paxos-based WORs (paxos-WOR) and Chain Replication WORs (chain-WOR), and compare it with CorfuDB [58], an unverified open source Java implementation of CORFU. Note that the WormLog code does not change for Chain Replication WORs (in fact, neither does the WormSpace stack above the WOR abstraction). However, performance differs due to $N+1$ RTT for the Chain Replication design and 2 RTT for the Paxos-based design. CorfuDB employs the same Chain Replication design as chain-WOR.

Figure 11 shows that with three wormservers, the write latency of a WormLog over paxos-WOR is the half of that for WormLog over chain-WOR for almost identical throughput. Under the same configuration, CorfuDB performs with 2 to 4X higher latency and 14% of the throughput of WormLog partly due to different languages for the implementation. We further vary the number of wormservers (replicas) and measure the access latency (Figure 12). While the Paxos-based WormLog has the same latency distribution regardless of the number of wormservers, Chain-Replication-based designs show linearly increasing latency with wider distributions depending on the number of wormservers. The experiment demonstrates that a Paxos-based WormSpace can enable a CORFU sequencer-based design while eliminating the latency of Chain Replication. Also, we show that different WOR implementations can be used without application code changes.

7.4 WormTX

Next, we present the performance of WormTX that implements fault-tolerant atomic commit. We compare the variants A8, B6, C5, D4, and E3, where A8 is the baseline that is equivalent to using Paxos as a black box to replicate the decision of resource managers. The numeric suffix represents the message delays of each WormTX variant. We focus on the latency as each variant optimizes for fewer message delays. Figure 13 illustrates linearly decreasing latency as more optimizations are applied. This shows that WormSpace facilitates applying optimizations step by step to the atomic commit protocol, and with the concurrency control mechanism described in Section 4.3, a low latency distributed transaction protocol can be easily supported on top of WormSpace.

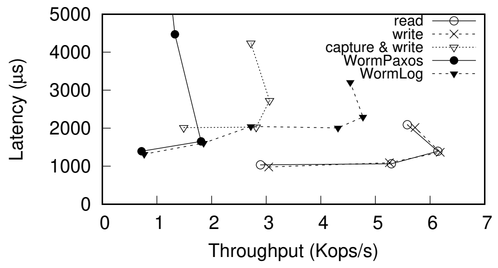

Figure 14: *Wormspace on CertiKOS: an end-to-end verified system is bottlenecked by the lwIP network stack.*

7.5 End-to-end Verification

Finally, we show the evaluation of WormSpace on CertiKOS which forms an end-to-end verified distributed system from the OS layer. To utilize CertiKOS, the experiment is run on a local cloud. Virtual machines are configured to mimic the set up in Amazon EC2: CertiKOS and WormSpace were placed inside QEMU instances with the same amount of resources as the m4.xlarge instance and the instances are placed such that all network communication crosses the physical machine boundaries.

We evaluate microbenchmark, WormPaxos, and WormLog and the throughput is approximately 10x lower and the latency is approximately 2x higher than running the experiments on Linux in Amazon EC2 (Figure 14). The main cause of this performance degradation has little to do with verification and is mainly attributed to the network stack used in CertiKOS. CertiKOS uses rather slow lwIP [15], which is intended for embedded systems, as its network stack and a single dedicated thread multiplexes packets to and from applications. The performance number showed similar results even when we ran all WormSpace servers and clients in a single VM due to this inefficiency. Once we replaced the network stack with a custom IPC call, we achieved over 100 Kops/s for all experiments when the same number of WormSpace clients and servers were placed in a single VM. We plan to replace lwIP with a higher-performance network stack for a better end-to-end performance in the future.

8 RELATED WORK

Distributed systems: A number of abstractions similar to the WOR exist in theoretical distributed systems, including sticky registers [48], consensus objects [29], and the Paxos register [37]; these are abstractions for theoretical reasoning. However, we propose the WOR as a programming abstraction and build a system exposing the WOR APIs. Other theoretical work points out the link between fault-tolerant atomic commit and consensus [18, 26]. Single writer many reader registers, which can be written multiple times, can be used to implement a WOR using a protocol like Disk Paxos [19]. Horus [56] is a modular stack for group communication that led to a verification effort called Ensemble [39].

Distributed applications often use services that embed consensus or replication protocols, such as Chubby [9] and Zookeeper [30]. WormSpace supports a more primitive abstraction compared to these services. Distributed transaction systems [44, 61] often combine transaction and consensus protocols, 'opening the Paxos box' to implement optimizations. These could conceivably be implemented over the WOR in similar fashion to the optimizations in Section 4.3.

Verification: Applying machine-checkable formal verification to real-world systems has been actively explored in recent years. IronFleet [27] and Verdi [59, 60] propose distributed system verification approaches and use Multi-Paxos/Raft as a verification target. IronFleet separates the verification into implementation, specification, and protocol layers; the first two layers are similar to a single WormSpace layer, and the protocol layer is similar to the WormSpace ghost layer. Verdi focuses on writing and verifying system code under an idealized network model first, and then adapting the proofs to a more realistic network model, whereas we assume an unreliable network to begin with. While both papers propose a systematic way to verify standalone distributed systems, WormSpace enables extensible verification via a modular layer-based verification approach, where the proofs can be reused and connected with new verified application layers.

It is well known that modularity leads to ease of verification. DISEL [50] verifies independent distributed protocols in isolation and horizontally combines them. Taube et al. [53] explores modularity for automated distributed system verification. Prior work has examined a layered storage system verification for crash safety [2, 11, 51] and a modular Paxos verification [7, 20, 47]. WormSpace shares the same insight about modularity, but leverages contextual refinement to provide incremental and extensible verification; enables both vertical and horizontal composition of layers; and verifies correctness of practical C programs in a concurrent and distributed environment.

Formal verification plays a key role for guaranteeing the correctness of security features [8, 16, 28, 41]. While WormSpace's proof does not focus on security, adding security features to the system and guaranteeing the security properties across WormSpace and application layers is a direction for future work.

WormSpace uses the CCAL approach [23, 25] for the verification. While CertiKOS [24] demonstrated the power of CCAL by verifying an entire OS, WormSpace augments the CCAL framework with an asynchronous network model to verify distributed systems and to connect verified distributed systems, applications and the OS.

9 CONCLUSION

Distributed systems are difficult to design, implement, and verify due to asynchrony and failures. Often, they re-implement the logic for consensus, durability, and availability in slightly different ways. The WOR abstraction proposed in this paper is the least common denominator for strongly consistent, fault-tolerant distributed systems. When the abstraction is exposed as a first-class programming primitive, it enables application stacks that are simple, realizing complex functionality in 100s of lines of code; flexible, allowing for different combinations of high-level application APIs and low-level consensus protocols; and verifiable, enabling layered verification techniques that allow easy and extensible verification of distributed application code.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful reviews and Cristina Nita-Rotaru for shepherding the paper. This work is funded and supported by NSF grants CCF-1637385, CCF-1521523, CCF-1763399, and CNS-1715154.

REFERENCES

[1] Rakesh Agrawal, Michael J Carey, and Larry W McVoy. 1987. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering* 12 (1987), 1348–1363.
[2] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C. Arpac-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Beyond storage APIs: provable semantics for storage stacks. In *USENIX Conference on Hot Topics in Operating Systems*. 20–20.
[3] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D Davis. 2012. CORFU: a shared log design for flash clusters.. In *USENIX Symposium on Networked Systems Design and Implementation*. 1–14.
[4] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: distributed data structures over a shared log. In *ACM Symposium on Operating Systems Principles*. 325–340.
[5] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder - a transactional record manager for shared flash. In *Biennial Conference on Innovative Data Systems Research*. 9–12.
[6] Kenneth P Birman. 1993. The process group approach to reliable distributed computing. *Commun. ACM* 36, 12 (1993), 37–53.
[7] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. 2003. Deconstructing Paxos. *SIGACT News* 34, 1 (2003), 47–67.
[8] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. 2017. Vale: verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*. 917–934.
[9] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation*. 335–350.
[10] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*. 173–186.
[11] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using crash Hoare logic for certifying the FSCQ file system. In *ACM Symposium on Operating Systems Principles*. 18–37.
[12] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*. 177–190.
[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.
[14] The Coq development team. 2018. The Coq proof assistant. http://coq.inria.fr.

[15] Adam Dunkels. 2001. *Design and implementation of the lwIP TCP/IP stack*. Technical Report. Swedish Institute of Computer Science.

[16] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: using verification to disentangle secure-enclave hardware from software. In *ACM Symposium on Operating Systems Principles*. 287–305.

[17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An empirical study on the correctness of formally verified distributed systems. In *European Conference on Computer Systems*. 328–343.

[18] Svend Frolund and Rachid Guerraoui. 2001. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems* 12, 2 (2001), 133–146.

[19] Eli Gafni and Leslie Lamport. 2003. Disk Paxos. *Distributed Computing* 16, 1 (2003), 1–20.

[20] Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos consensus, deconstructed and abstracted. In *European Symposium on Programming*. 912–939.

[21] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems* 31, 1 (2006), 133–160.

[22] James N Gray. 1978. Notes on data base operating systems. In *Operating Systems: An Advanced Course*. Springer, 393–481.

[23] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 595–608.

[24] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *USENIX Conference on Operating Systems Design and Implementation*. 653–669.

[25] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 646–661.

[26] Vassos Hadzilacos. 1990. On the relationship between the atomic commitment and consensus problems. In *Fault-Tolerant Distributed Computing*. Springer, 201–208.

[27] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *ACM Symposium on Operating Systems Principles*. 1–17.

[28] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad apps: end-to-end security via automated full-system verification. In *USENIX Conference on Operating Systems Design and Implementation*. 165–181.

[29] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (1991), 124–149.

[30] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, Vol. 8. 9.

[31] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: high-performance broadcast for primary-backup systems. In *IEEE/IFIP International Conference on Dependable Systems Networks*. 245–256.

[32] Jieung Kim, Vilhelm SjÃűberg, Ronghui Gu, and Zhong Shao. 2017. Safety and liveness of MCS lockâĂŤlayer by layer. In *Asian Symposium on Programming Languages and Systems*. 273–297.

[33] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

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

[35] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical Paxos and primary-backup replication. In *ACM Symposium on Principles of Distributed Computing*. 312–313.

[36] K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. 348–370.

[37] Harry C Li, Allen Clement, Amitanand S Aiyer, and Lorenzo Alvisi. 2007. The Paxos register. In *IEEE International Symposium on Reliable Distributed Systems*. IEEE, 114–126.

[38] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 455–468.

[39] Xiaoming Liu, Christoph Kreitz, Robbert Van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. 1999. Building reliable, high-performance communication systems from components. In *ACM Symposium on Operating Systems Principles*. 80–92.

[40] Joshua Lockerman, Jose M Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: a partially ordered shared log. In *USENIX Symposium on Operating Systems Design and Implementation*. 357–372.

[41] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. 2013. Verifying security invariants in ExpressOS. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 293–304.

[42] Dahlia Malkhi, Mahesh Balakrishnan, John D Davis, Vijayan Prabhakaran, and Ted Wobber. 2012. From Paxos to CORFU: a flash-speed shared log. *ACM SIGOPS Operating Systems Review* 46, 1 (2012), 47–51.

[43] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *ACM Symposium on Operating Systems Principles*. 358–372.

[44] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating concurrency control and consensus for commits under conflicts. In *USENIX Conference on Operating Systems Design and Implementation*. 517–532.

[45] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: push-button verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*. 252–269.

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

[47] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 108:1–108:31.

[48] S. A. Plotkin. 1989. Sticky bits and universality of consensus. In *ACM Symposium on Principles of Distributed Computing*. 159–175.

[49] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[50] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 28:1–28:30.

[51] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-button verification of file systems via crash refinement. In *USENIX Conference on Operating Systems Design and Implementation*. 1–16.

[52] Andrew S. Tanenbaum and Maarten van Steen. 2006. *Distributed systems: principles and paradigms (2nd edition)*. Prentice-Hall, Inc.

[53] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability: implementing and semi-automatically verifying distributed systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 662–677.

[54] Jeff Terrace and Michael J Freedman. 2009. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*. 11–11.

[55] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *Comput. Surveys* 47, 3 (2015), 42.

[56] Robbert Van Renesse, Kenneth P Birman, and Silvano Maffeis. 1996. Horus: a flexible group communication system. *Commun. ACM* 39, 4 (1996), 76–83.

[57] Robbert Van Renesse and Fred B Schneider. 2004. Chain replication for supporting high throughput and availability.. In *USENIX Conference on Operating Systems Design and Implementation*. 91–104.

[58] VMware Research. 2018. CorfuDB. https://www.github.com/CorfuDB/CorfuDB.

[59] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 357–368.

[60] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *ACM SIGPLAN Conference on Certified Programs and Proofs*. 154–165.

[61] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *ACM Symposium on Operating Systems Principles*. 263–278.