Write-Once Registers:
A Modular Foundation for Simple, Verifiable Distributed Systems

Ji-Yong Shin† Jieung Kim† Wolf Honore† Hernán Vanzetto†
Srihari Radhakrishnan§ Mahesh Balakrishnan‡ Zhong Shao†

†Yale University §Duke University ‡Facebook

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 or surpass the performance of unverified monolithic implementations. We use a modular layered verification approach to link the verification proofs for the applications, WormSpace, and a verified OS to produce an end-to-end verified distributed system stack from the application to the OS.

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) [23] 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 [33], 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 [45, 51], shared logs [4], groups [7, 52], namespaces [10, 31], filesystems, databases [6], or objects [5]. 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 fault-tolerant protocols, each decision WOR is either layered inefficiently over a replicated state machine, or entwined with a transaction coordination logic [22].

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 [24, 26], 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 (contrasted from Write-Once-Read-Many Address Space), which provides applications with a shared address space of durable, highly available, and strongly consistent WORs (see Figure 1). WormSpace di-
vides 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 [15] 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/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 append-only 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) [24], 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 [25], a fully verified OS, enabling a 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 [18]. 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 or surpassing 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 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.

Shared logs 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 writeonce 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 in-
operation. As with SMR, the GC send/receive API can be
implemented via an orthogonal mechanism such as locking.

Coordination services (e.g., Chubby and ZooKeeper) typi-
ically expose a filesystem-like API to applications. Such
an API is ideal for use cases such as membership manage-
ment and leadership election, but is awkward at best 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 coor-
динating transactions across distributed state. In effect, they are
manipulating a set of WORs, each one representing the pre-
pare/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 denomi-

ator interface: all the systems described above can be im-
plemented easily and efficiently over a WOR.

2.2 Verification Approach

Modularity of WOR enables verification based on the certi-
ified concurrent abstraction layer (CCAL) approach [24, 26],
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 be-
tween layers. Each layer \( L \) is a state machine which has
its corresponding implementation \( i \) and an execution envi-
ronment 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 distri-
buted system. Informally, a layer \( L_{\text{low}} \) contextually re-
fines the higher layer \( L_{\text{high}} \) if each state transition made by
\( L_{\text{high}} \) based on any context \( t \) corresponds to a sequence of
state transitions by \( L_{\text{low}} \) which has the context \( t \) and \( L_{\text{high}} \)’s
implementation \( i_{\text{high}} \). We can formally represent context-
ual refinement \( L_{\text{low}} \subseteq_{cr} L_{\text{high}} \) as verifying the follow-
ing:
\[
\forall t, L_{\text{low}}(i_{\text{high}} \oplus t) \subseteq L_{\text{high}}(t),
\]
where \( \subseteq \) 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 contextually refines the layer
above it for an arbitrary context. When the stack is ex-
tended with a new verified layer on top, the inter-layer con-
textual refinement proofs can be reused with an updated con-
text to include the new layer. For example, if we add a new layer \( L_{\text{top}} \) on top of verified layers \( L_{\text{mid}} \) and \( L_{\text{btm}} \),
we need one new proof that shows \( L_{\text{mid}} \) contextually re-
fines \( L_{\text{top}} \), but we can reuse the proof for \( L_{\text{btm}} \subseteq_{cr} L_{\text{mid}} \) without requiring any modification to the proof because
the proof holds “for all” context \( t \). After the proof of
\( L_{\text{mid}} \subseteq_{cr} L_{\text{top}} \), we are automatically guaranteed that \( L_{\text{btm}} \)
contextually refines all the way up to \( L_{\text{top}} \) as follows:
\[
\forall t, L_{\text{btm}}(i_{\text{mid}} \oplus (i_{\text{top}} \oplus t)) \subseteq L_{\text{mid}}(i_{\text{top}} \oplus t) \subseteq L_{\text{top}}(t).
\]

Internally, each layer is composed of the C implemen-
tation, 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 [24]; 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 \n

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 represen-
tation) 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 from \( L_{\text{btm}} \) to \( L_{\text{top}} \)
function correctly independently and together. With the help
of the verified CompCertX compiler, the correctness of sys-
tem 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 \( L_{\text{top}} \). The application uses \( L_{\text{top}} \) as its bottom layer for
verification and is oblivious to the layers underneath. The con-
textual refinement relation between \( L_{\text{top}} \) and the application
guarantees that the application uses the underlying system
(from \( L_{\text{btm}} \) to \( L_{\text{top}} \)) correctly.

Such co-verification of the application and the system is
critical, but often overlooked and considered difficult. With-
out co-verification, the application and the system can be ver-
ified independently but still be incorrect as a whole, since
the application can abuse the system interface or take actions
based on wrong assumptions [18]. For example, for the same
write interface, the system and the application may have
different address bounds and the application can write be-


on the system’s address limit. Another example involves


storage system to mean “stored on a backup machine”; both


mismatches can neutralize the verification effort. Con-
textual 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.

In addition to the functional correctness proof, we verify
// allocates a WOS
int WS_alloc(char *metadata, int size, segno *newsegno);

// trims a segment
int WS_trim(segno t seg);

// batch captures a sub-range within the WOS
int WOS_capture(segno t seg, int *retcodes, off t start, off t end);

// batch writes a sub-range within the WOS
int WOS_write(segno t seg, char *buf, int size, int *retcodes, off t start, off t stop);

// registers a listener for write notifications
int WOS_listen(segno t seg, callback_t listener);

// captures a WOR
int WOR_capture(segno t seg, off t addr, int *captureID);

// writes a single WOR
int WOR_write(segno t seg, off t addr, char *buf, int size, int captureID);

// reads a single WOR
int WOR_read(segno t seg, off t addr, char *buf, int size);

Figure 2: The WormSpace API.

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 a capture and write API, which act as batched or vectorized 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 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 subsequently 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). 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 (one per address in that segment), 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 [4]. 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 [50] 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 [11, 13]; we leave this for future work.

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 MultiPAXOS 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 MultiPAXOS (in the same way that a block device can be implemented over a filesystem). However, the more conventional layering places the WOR at the bottom of the stack to simplify higher-level systems, as we now describe.

# 4 WormSpace Applications

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

---

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

**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 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 in the WormSpace; and learn commands by reading the address space in sequential order. If a proposing client finds that the current tail is at the end of a WOS, it allocates a new one and then writes to the next address.

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**: 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, 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 zero modification over a WOR implementation based on Chain Replication rather than Paxos; in contrast, existing Multi-PAXOS implementations require extensive modification to run over a different single-shot consensus protocol.

**Flexible Leadership**: Sticky leadership – i.e., retaining a single leader across multiple commands – is a key performance imperative for Multi-PAXOS implementations, since it A) allows commands to be decided within a single round-trip rather than two in the absence of failures, and B) eliminates contention between leaders. In many Multi-PAXOS implementations, leadership strategy is baked into the system design; for example, Raft [43] 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; ac-
Figure 4: WormLog: clients can append by obtaining a token from the sequencer and writing to WormSpace.

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.

**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 information can be piggy-backed 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 [51]).

**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. 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 [4] 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 [39]: 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; 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) [48] 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 to using Paxos as a black-box.

[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.

[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.

[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 [22]; 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 the protocol 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 [48]. Instead, we assemble a non-blocking protocol via simple, intuitive, and data-centric commands on WORs.

Concurrency Control: The proposed atomic commit schemes can be integrated with concurrency control schemes based on timestamps, deadlock detection, etc. We implemented a simple concurrency control protocol based on locking that uses Immediate-Restart [2] 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 immediately 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 [24, 26] introduced in Section 2.2, modeling an asynchronous network of distributed nodes in order to verify WormSpace. We apply CCAL beyond a single system verification for the first time and link the proof of WormSpace, 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 horizontally composes the two stacks and proves properties across multiple wormservers and clients. The ghost layer includes a global state transition system that can reason all concurrent client and server interactions based on a network model. Safety properties of Paxos 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:

```plaintext
Function 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
  WOR_write segment offset val cid adt.
```

We verify the WOR abstraction, the WOS abstraction, and the WormSpace API. 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 [26]. Our network model 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).

Network communication patterns can be complex when a client interacts with multiple wormservers in a one-to-many request pattern. Abstraction and contextual refinement can help us manage this complexity without reducing the fidelity of verification. Accordingly, we create a network log layer with simpler semantics, and prove that the original log refines the simplified log. The simplified log coalesces broadcasts and receptions into singleton events and eliminates duplicates simplifying global property proofs.

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
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\_L := (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\_L := bcast\_n\_recv nid addr val cid net\_L 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\_L))\_2)[nid]] addr in
(* log the result using a ghost msg to the net log *)
let net\_L := (ghost\_result nid result) := net\_L adt in
(* return the updated net log and the result *)
(adt {net\_L := net\_L}, result)
else None.

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

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).

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/immutability 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 following key lemma:

**Lemma 1.** Given a valid network log \( \ell \), 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 \( \ell \)
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 for
WormPaxos/WormLog) and WOS trim are easily verified.

5.4 Top-Level Theorem of WormSpace

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

**Theorem 2.** \( \forall t, L\_TCB(i\_All\_WormSpace \oplus t) \subseteq L\_WormSpace(t) \),
where \( t \) is the context and \( i\_All\_WormSpace \) 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 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 refine-
ment 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.** \( \forall t, L\_WormSpace(t) \subseteq L\_WormApp(t) \),
where WormApp can be one of WormPaxos, WormLog,
and WormTX. By reusing Theorem 2 and transitively combining it
with Theorems 3, applications are guaranteed to be cor-
correct with respect to all layers of WormSpace and to encapsu-
late verified Paxos properties. Similarly, Theorem 2 can be
reused to verify any system in Section 2.1 to guarantee WOR
semantics, if we use WormSpace as a building block.

To enable end-to-end verification of WormSpace, Worm-
Paxos, 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 co-
designed, 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 (\( L\_\text{os} \)) of the OS without any
side-effects [18]; this verifies and guarantees,

**Theorem 4.** \( \forall t, L\_\text{os} \subseteq L\_\text{CertiKOS} \oplus L\_\text{WormSpace} \oplus L\_\text{WormApp} \oplus t \)
\( \subseteq L\_\text{WormApp}(t) \).

The extensibility of WormSpace verification to applica-
tions and the OS is difficult for other verified systems [28,
42] 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 (e.g., Dafny [36] and Z3 [14]).
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 is fully verified (either machine checked or hand proven), in contrast to other work that often includes some untrusted component (e.g., to generate executable code). A small part of our end-to-end system remains unverified: the listen function of the WOS; the reconfiguration protocol in WormSpace; and the Chain Replication WOR. We did not verify application-level concurrency within a single process, while concurrent processes are verified correct against WormSpace. CCAL supports liveness proofs [32], but we left them as future work.

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.

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 end-to-end verification can be extended easily (e.g., a key-value store layer above WormPaxos), 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 fast. We run 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 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 wormserver. 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 former takes 2 RTT whereas the latter takes 1 RTT to complete the write. The overhead of incorporating a capture call 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 [40]. 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/C-

<table>
<thead>
<tr>
<th>WormSpace</th>
<th>WormPaxos</th>
<th>WormLog</th>
<th>WormTX (C5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4551</td>
<td>359</td>
<td>362</td>
<td>547</td>
</tr>
</tbody>
</table>

Table 1: C lines of code (CLoC) for WormSpace and applications. C5 has the largest CLoC among WormTX variants.

Technical Report YALEU/DCS/TR1544 December 18, 2018
WormPaxos 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. Our point here is not to claim WormPaxos simply runs much faster than EPaxos, which internally does dependency checks and ordering, but to show that verified code is not necessarily slow and can be even faster than unverified code.

We also measure the throughput with data persistence on an Amazon EBS GP2 SSD (Figure 10). Having to write less data to persistent storage, due to the absence of a commit phase, makes WormPaxos achieve higher throughput.

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 [1], 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: the Chain Replication design propagates the data from the head server to the tail server in a sequence before returning a write; this incurs 1 RTT per wormserver in the chain. In addition to contacting wormservers, clients contact the sequencer before issuing a write. Thus, having N wormservers results in N + 1 RTT for the Chain Replication design and 2 RTT for the Paxos-based design (wormservers are accessed in parallel). 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. We further vary the number of wormservers (replicas) and measure the access latency. 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 (Figure 12). 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 and both WormLogs outperform CorfuDB partly due to different languages for the implementation.

7.4 WormTX

Next, we present the performance of WormTX that implements fault-tolerant atomic commit. We compare the commit latency of WormTX variants A8, B6, C5, D4, and E3. The numeric suffix represents the message delays of each WormTX variant. Figure 13 illustrates linearly decreasing latency as more optimizations are applied. This shows that a low-latency distributed transaction protocol can be easily implemented above WormSpace.

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. The experiment was run on a local cloud where the virtual machine configuration mimics 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 [16], 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 [44], consensus objects [30], 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 [19, 27]. Single writer many reader registers, which can be written multiple times, can be used to implement a WOR using a protocol like Disk Paxos [20].

Distributed applications often use services that embed consensus or replication protocols, such as Chubby [10] and Zookeeper [31]. WormSpace supports a more primitive abstraction compared to these services. Distributed transaction systems [41, 55] 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 [28] and Verdi [54] 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 [46] verifies independent distributed protocols in isolation and horizontally combines them. Taube et al. [49] explores modularity for automated distributed system verification. Prior work has examined a layered storage system verification for crash safety [3, 12, 47] and a modular Paxos verification [8, 21]. 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 based programs in a concurrent and distributed environment.

Formal verification plays a key role for guaranteeing the correctness of security features [9, 17, 29, 38]. 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 same CCAL approach [24, 26] as CertiKOS [25]. While CertiKOS demonstrated the power of CCAL by verifying an entire OS, WormSpace showed that CCAL can be extended to connect verified systems and applications and to model and verify distributed systems on an asynchronous network.

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 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, extensible verification of distributed application code.
References


