

# UpRight Cluster Services

Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang,  
Lorenzo Alvisi, Mike Dahlin, Taylor Riché  
Department of Computer Sciences  
The University of Texas at Austin  
Austin, Texas, USA  
{aclement, manos, sangmin, yangwang  
lorenzo, dahlin, riche}@cs.utexas.edu

## ABSTRACT

The UpRight library seeks to make Byzantine fault tolerance (BFT) a simple and viable alternative to crash fault tolerance for a range of cluster services. We demonstrate UpRight by producing BFT versions of the Zookeeper lock service and the Hadoop Distributed File System (HDFS). Our design choices in UpRight favor simplifying adoption by existing applications; performance is a secondary concern. Despite these priorities, our BFT Zookeeper and BFT HDFS implementations have performance comparable with the originals while providing additional robustness.

## Categories and Subject Descriptors

C.2.4 [Computer Systems Organization]: Distributed Systems—*Client/server*; D.4.5 [Operating Systems]: Reliability—*Fault-tolerance*

## General Terms

Design, Reliability

## Keywords

Byzantine fault tolerance, Cluster services, Reliability

## 1. INTRODUCTION

Our objective is to make Byzantine fault tolerance (BFT) something that practitioners can easily adopt both to safeguard availability (keeping systems up) and to safeguard correctness (keeping systems right.) To that end, we construct UpRight, a new library for fault tolerant replication, and we use it to build BFT versions of two widely-deployed open-source crash fault tolerant (CFT) systems, the Zookeeper coordination service [35] and the Hadoop Distributed File system (HDFS) [16].

Practitioners routinely pay non-trivial costs to tolerate crash failures (e.g., off-line backup, on-line redundancy [10, 15], Paxos [6, 20, 31]). However, although non-crash failures

occur with some regularity and can have significant consequences [2, 7, 30] and although the research community has done a great deal of work to improve BFT technologies [1, 8, 11, 12, 18, 19, 32–34], deployment of BFT replication remains rare.

We believe that for practitioners to see BFT as a viable option they must be able to use it to build and deploy systems of interest at low incremental cost compared to the CFT systems they build and deploy now: BFT systems must be competitive with CFT systems not just in terms of performance, hardware overheads, and availability, but also in terms of engineering effort.

With respect to the first three factors, recent research has put many of the pieces in place—performance can be excellent [1, 8, 11, 12, 18, 19, 32–34], replication costs low [33, 34], and robustness good [5, 11]. The UpRight library draws heavily on this work to retain these properties.

With respect to engineering effort, to be truly low cost, BFT must mesh well with large existing code bases such as HDFS and Zookeeper. Unfortunately, the current state of the art often requires rewriting applications from scratch. If the cost of BFT is “rewrite your cluster file system,” then widespread adoption will not happen. Our design choices in UpRight favor minimizing intrusiveness to existing applications over raw performance.

We construct UpRight-Zookeeper and UpRight-HDFS using the Zookeeper and HDFS open source code bases. Both resulting systems provide potentially interesting improvements in fault tolerance. Whereas the existing HDFS system can be halted by a single fail-stop node, UpRight-HDFS has no single points of failure and also provides end-to-end Byzantine fault tolerance against faulty clients, DataNodes, and NameNodes. Similarly, although Zookeeper guards against fail-stop faults, a data center typically runs a single instance of a coordination service on which a wide range of cluster services depend [9], so it may be attractive to invest modest additional resources in this critical service to protect against a wider range of faults.

In both cases, the cost of BFT over CFT is low in key dimensions. Although attaching these applications to the UpRight library is not automatic, it is straightforward. With respect to performance, as Figure 1 highlights, despite our design choices minimizing intrusiveness on existing code, the BFT systems are competitive with the original ones. However, the overheads of computing cryptographic digests on messages do cause the BFT systems to consume more CPU cycles than the original systems at a given level of load.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

Copyright 2009 ACM 978-1-60558-752-3/09/10 ...\$10.00.

System	Workload	Original	UpRight
Zookeeper	90/10 read/write	15,645 Ops/s	10,823 Ops/s
HDFS	Read	746.38 MB/s	716.52 MB/s
	Write	237.78 MB/s	166.91 MB/s

**Figure 1: Throughput for original and UpRight Zookeeper and HDFS systems. We detail our experiments in Sections 5 and 6.**

The contribution of this paper is to establish Byzantine fault tolerance as a viable alternative to crash fault tolerance for at least some cluster services rather than any individual technique. Much of our work involved making existing ideas fit well together, and in the sections that follow, we highlight the lessons learned from these engineering efforts and also highlight where we found that new techniques were needed. We do not claim that all cluster services can get low-cost BFT. We merely provide evidence that some interesting services can, and we provide a library and experience that may help others do so.

The paper proceeds as follows:

- |                         |                 |
|-------------------------|-----------------|
| 1. Introduction         | 5. Zookeeper    |
| 2. Model                | 6. HDFS         |
| 3. UpRight applications | 7. Related work |
| 4. UpRight agreement    | 8. Conclusions  |

## 2. MODEL

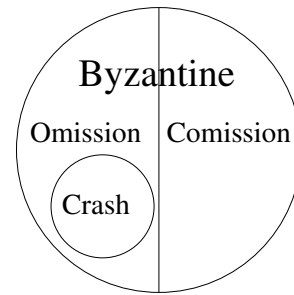
UpRight makes standard assumptions for Byzantine fault tolerant systems with a few tweaks and clarifications.

**Standard assumptions.** We assume the Byzantine failure model where some faulty nodes (servers or clients) may behave arbitrarily [22]. We assume a strong adversary that can coordinate faulty nodes to compromise the replicated service. We do, however, assume the adversary cannot break cryptographic techniques like collision-resistant hashes, encryption, and signatures.

Our system’s safety properties hold in any asynchronous distributed system where nodes are connected by a network that may fail to deliver, corrupt, delay, or reorder messages. Liveness, however, is guaranteed only during *synchronous intervals* in which messages sent between correct nodes are processed within some fixed (but potentially unknown) worst case delay from when they are sent.

**Tweak: Number of failed nodes.** Under the Byzantine fault model, systems are typically designed to tolerate  $t$  Byzantine failures of any kind. Instead, we allow UpRight to be configured to tolerate a larger number of nodes that fail by omission (e.g., crashing) than nodes that fail by commission (e.g., taking incorrect actions).

Configuring separate fault tolerance thresholds for omission and commission failures is beneficial for three reasons. (1) Omission failures are likely to be more common than commission failures. (2) Configuring separate fault tolerance thresholds allows us to build systems that match typical commercial deployment goals with respect to omission tolerance and add incremental commission tolerance at incremental cost. (3) Tolerating a small number of commission failures may capture the sweet spot for Byzantine fault tolerance in cluster environments: tolerating one or a few



**Figure 2: Failure hierarchy. We divide the space of arbitrary Byzantine failures into omission failures and commission failures. Crash failures are a subset of omission failures.**

uncorrelated failures (e.g., lost disk sectors) will usually be enough, and when it is not, there may be no feasible way to tolerate highly-correlated failures such as software bugs that allows a malicious intruder to control all replicas.

Formally, as Figure 2 illustrates, we divide the space of arbitrary *Byzantine* failures into *omission* failures (including crash failures) in which a node fails to send one or more messages specified by the protocol and sends no incorrect messages based on the protocol and its inputs and *commission* failures, which include all failures that are not omission failures, including all failures in which a node sends a message that is not specified by the protocol.

We design UpRight to provide the following properties

- An UpRight system is safe (“right”) despite  $r$  commission failures and any number of omission failures.
- An UpRight system is safe and eventually live (“up”) during sufficiently long synchronous intervals when there are at most  $u$  failures of which at most  $r$  are commission failures and the rest are omission failures.<sup>1</sup>

For example, if  $u = 3$  and  $r = 1$  the system can operate with 3 crashed nodes or two crashed nodes and one node whose data structures become corrupted causing it to send erroneous responses to requests. Notice that configurations in which  $u = r$  are equivalent to the customary formulation of the Byzantine failure model and configurations and configurations with  $r = 0$  are equivalent to the customary formulation of the crash failure model.

**Clarification: Crash-recover incidents.** UpRight is designed to tolerate any number of nodes that crash and recover. In a crash-recover incident, a process loses its volatile state, may fail to send one or more required messages, but eventually resumes operation in a state consistent with its state prior to crashing.

Formally, nodes that crash and recover count as suffering an omission failure during the interval they are crashed and count as correct after they recover.

<sup>1</sup>More generally, the system is eventually live—it outputs a response to an input—during sufficiently long synchronous intervals despite  $u$  failures of any type. However, this response may be incorrect if more than  $r$  of the failures are commission failures. Also note that for simplicity throughout this paper we assume  $u \geq r$ .

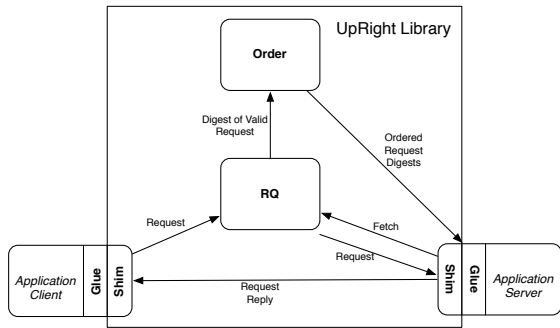


Figure 3: UpRight architecture.

In an asynchronous system, crash/recover nodes are often modeled as correct, but temporarily slow, rather than failed. We explicitly account for crash-recovery incidents because (1) temporarily failed nodes affect the liveness guarantees our system can provide and (2) ensuring safety despite widespread crashes (e.g., due to a power outage) raises engineering issues that we discuss in Section 4.2.

**Tweak: Robust performance.** For our target application space of critical cluster services, a weak liveness guarantee such as “Eventually the system makes progress” is insufficient. It would not be acceptable to design a system whose throughput falls by orders of magnitude when a single fault occurs. We must ensure not only that our system is *eventually live* during synchronous intervals, but that it delivers *good performance* during *uncivil intervals*—synchronous intervals during which a bounded number of failures occur [11].

We therefore impose the following requirement

- An UpRight system ensures safety and good performance during sufficiently long synchronous intervals with an application-dependent bound on message delivery when there are at most  $u$  failures of which at most  $r$  are commission failures.

### 3. UPRIGHT APPLICATIONS

UpRight implements state machine replication [28], and it tries to isolate applications from the details of the replication protocol in order to make it easy to convert a CFT application into a BFT one or to construct a new fault tolerant application. The author of an UpRight application needs to know the interface provided by the UpRight library and the details of how the application processes requests and manages its state; the author does not need to know about the details of fault tolerance or replica coordination.

From the application’s perspective, the architecture of an UpRight system is simple as shown in Figure 3. A client sends requests to and receives replies from the UpRight library, and a server does the reverse; a server also exposes checkpoints of its local state to the library. More specifically, each client or server node is composed of the application itself, which accesses the local UpRight *shim* via application-specific *glue*. The UpRight shim is generic and handles communication with the other UpRight components. The application-specific glue bridges the gap between a legacy application’s code and the interface exported by the UpRight shim; new applications may be written directly to the

shim interface and dispense with a separate glue layer entirely.

The UpRight library ensures that each application server replica sees the same sequence of requests and maintains consistent state, and it ensures that an application client sees responses consistent with this sequence and state. Ensuring that responses and state are consistent raises specific challenges that applications must address in request execution and checkpoint management.

### 3.1 Request execution

UpRight applications follow a basic client-server architecture in which a client issues a request and expects a response from the server. When processing a sequence of requests, every server is expected to provide the same sequence of responses. In order to accomplish this goal, applications must account for nondeterminism, multithreaded execution, read only requests, and spontaneous server-initiated replies.

**Nondeterminism.** Many applications rely on real time or random numbers as part of normal operation. These factors can be used in many ways including garbage collecting soft state, naming new data structures, or declaring uncommunicative nodes dead. Each request issued by the UpRight shim to the application server glue is accompanied by a time and random seed to be used in conjunction with executing the request [8]. UpRight applications must be modified to rely on these specified times rather than the local machine time and to use the random seed as appropriate when using a pseudo random number generator. An alternative to modifying the application is to modify the runtime system as is done in model checking frameworks [17].

**Multithreading.** Parallel execution allows applications to take advantage of hardware resources, but application servers must ensure that the actual execution is equivalent to executing the requests sequentially in the order specified by the UpRight library. The simplest way to enforce this requirement is for the glue to complete execution of request  $i$  before beginning execution of request  $i + 1$ . More sophisticated glue may issue independent requests in parallel [19, 32].

Some systems include “housekeeping” threads that asynchronously modify application server state. For example, an HDFS server maintains a list of live data servers, removing an uncommunicative server from the list after a timeout. An application must ensure that housekeeping threads run at well-defined points in the sequence of requests by, for example, scheduling such threads at specific points in virtual time rather than at periodic real time intervals.

**Read only replies.** As a performance optimization, UpRight supports PBFT’s read-only optimization [8], in which a client shim sends read-only, side-effect-free requests directly to the server shims and server shims execute them without ordering them in the global sequence of requests. If a quorum of replies match, the client can use the reply; otherwise the request is concurrent with an interfering operation, and the client shim must reissue the request via the normal path to execute the request in the global sequence of requests. To support this optimization, the client and server glue must identify read only requests.

In some cases, read-only replies may include rapidly changing state that is likely to differ across replicas but that can be canonicalized to a legal value at the client. For example, if a client receives  $2r + 1$  read-only replies that match in all fields except a real-time timestamp, a replica might map the replies to a canonicalized reply comprising the median timestamp and the other matching fields. Our client shim therefore passes read-only replies to the client glue to optionally do application-specific canonicalization.

**Spontaneous replies.** Applications may allow the server to push unsolicited messages to clients. For example, in Zookeeper, a client can *watch* an object and be notified of any changes to it. Unfortunately, in an asynchronous system with unreliable networks, there is no bound on the number of push messages sent by the server but not received by the client. UpRight must either require unbounded storage or provide the abstraction of an unreliable channel.

UpRight provides unreliable channels for push events. We posit that most crash fault tolerant systems will already cope with the “lost message” case (e.g., to handle the case when the TCP connection is lost and some events occur before the connection to the server can be reestablished), so if the client shim/glue activates this code whenever a message is lost, existing application semantics are preserved.

For example, in our Zookeeper case study, a client expects to see all updates on a watched file unless it sees a TCP connection break event, which causes it to reestablish the watches. The existing recovery code determines if any watched files changed while the connection was broken.

In our implementation, the server shim includes sequence numbers on push events, sends them in FIFO order, and attempts to resend them until they are acknowledged, but a node can unilaterally garbage collect any pending push events at any time. The client shim gathers quorums of push messages for each sequence number and signals the client glue if there is a gap in the sequence. The glue exposes such gaps via the (presumed existing) application lost-message or lost-connection handler.

## 3.2 Checkpoints

In an asynchronous system, even correct server replicas can fall arbitrarily behind, so BFT state machine replication frameworks must provide a way to checkpoint a server replica’s state, to certify that a quorum of server replicas have produced identical checkpoints, and to transfer a certified checkpoint to a node that has fallen behind [8]. Therefore, the UpRight server shim periodically tells the server application to checkpoint its state to stable storage; later, the shim asks the application to provide a cryptographic hash to identify that stable checkpoint state. If a replica falls behind, its server shim communicates with the other server shims to retrieve the most recent checkpoint, restarts the server application using that state, and finally replays the log of ordered requests after that checkpoint to bring the replica to the current state.

Server application checkpoints must be (1) inexpensive to generate because checkpoint frequency is relatively high, (2) inexpensive to apply because the replication framework uses checkpoints in both the rare case of a machine crashing and restarting and the more common case of a machine falling behind on message processing, (3) deterministic because correct nodes must generate identical checkpoints for a given

request sequence number, and (4) nonintrusive on the codebase because we must not require extensive modifications of applications.

There is tension among these requirements. For example, generating checkpoints more frequently increases generation cost but reduces recovery time (because the log that must be applied will be correspondingly shorter.) For example, requiring an application to store its data structures in a memory array checksummed with a Merkle tree [8] can reduce checkpoint generation and fetch time (since only changed parts need be stored or fetched) but may require intrusive changes to legacy applications.

UpRight therefore allows server applications and server glue to implement different checkpoint strategies. To simplify development, the UpRight library provides three checkpoint *glue libraries* that implement a hybrid checkpoint/delta approach and that provide three simple options for deterministically checkpointing application state: stop and copy, helper process, and copy-on-write. A given application’s glue can incorporate one of these existing libraries, or it can implement its own checkpoint strategy [8, 33].

**Hybrid checkpoint/delta approach.** The hybrid checkpoint/delta approach seeks to minimize intrusiveness to legacy code.

We posit that most crash fault tolerant services will already have some means to checkpoint their state. So, to minimize intrusiveness, to lower barriers to adoption, and to avoid the need for projects to maintain two distinct checkpoint mechanisms, we wish to use applications’ existing checkpoint mechanisms. Unfortunately, the existing application code for generating checkpoints is likely to be suitable for infrequent, coarse grained checkpoints. For example, both the HDFS and Zookeeper applications produce their checkpoints by walking their important in-memory data structures and writing them all to disk.

The hybrid checkpoint/delta approach uses existing application code to take checkpoints at the approximately the same coarse-grained intervals the original systems use. We presume that these intervals are sufficiently long that the overhead is acceptable. To produce the more frequent checkpoints required by the UpRight shim, the glue library augments these infrequent, coarse-grained, application checkpoints with frequent fine-grained deltas that comprise a log of requests from the previous delta or checkpoint to the next.

Within the hybrid checkpoint/delta approach, the application’s checkpoints must be produced deterministically. The three glue libraries support three options for doing so with different complexity/performance trade-offs.

**Stop and copy.** If an application’s state is small and an application can tolerate a few tens of milliseconds of added latency, the simplest checkpoint strategy is to pause the arrival of new requests so that the application is quiescent while it writes its state to disk. Since we eliminate other sources of nondeterminism as described above, this approach suffices to ensure that replicas produce identical checkpoints for a given sequence number.

**Helper process.** The helper process approach produces checkpoints asynchronously to avoid pausing request execution and seeks to minimize intrusiveness to legacy code.

To ensure that different replicas produce identical checkpoints without having to pause request processing, each node runs two slightly modified instances of the server application process—a primary and a helper—to which we feed the same series of requests. We deactivate the checkpoint generation code at the primary. For the helper, we omit sending replies to clients, and we pause the sequence of incoming requests so that it is quiescent while it is producing a checkpoint.

**Copy on write.** Rather than use a helper process to produce a deterministic checkpoint, applications can be modified so that their key data structures are treated as copy on write while checkpoints are taken [9]. This approach is more invasive but can have lower overheads than the helper-based approach. Typically, these changes involve adding a few fields to each object, modifying each method that modifies an object’s state to clone the object if a checkpoint is currently being taken and a clone has not yet been made, and modifying the serialization code to serialize the clone rather than the updated object if the clone was created during the current checkpoint epoch. We currently make these changes by hand, but we believe the process of modifying each class that implements the COW interface could be automated via source-to-source translation or binary modification.

**Discussion.** With respect to our goal of nonintrusiveness, our experience so far is positive. While it seems plausible that most CFT applications of interest will include the ability to checkpoint their own state in a deterministic way when they are quiescent, we did consider several alternatives.

Systems in the PBFT lineage [1, 8, 11, 12, 18, 19, 32–34] modify applications to store their checkpoint state in a special region of memory and to access that state via a special API so that the system can track which pages have been modified. The benefit of this approach is low incremental checkpoint generation cost. The disadvantage is more extensive modification of existing applications.

ZZ [33] uses the PBFT memory system to checkpoint an application’s in-memory state, but it also supports applications whose state includes on-disk files. ZZ uses file system snapshots to inexpensively create copy-on-write versions of those files, and it allows a server application node to quickly begin executing from the memory state checkpoint and fetch file system state in the background or on demand. These techniques are complementary to the ones we explore for checkpoints of in-memory state.

We also considered using OS support for fork to provide copy-on-write snapshots of application state or using OS support for memory system protection, but such techniques do not easily mesh with a Java application running in a JVM. Authors of other applications should consider these approaches for helping to produce deterministic snapshots without the performance costs of stop and copy or a helper process and without the invasiveness of application-level copy on write.

## 4. UPRIGHT AGREEMENT

Figure 3 illustrates UpRight’s high-level architecture. At UpRight’s core is a Byzantine agreement protocol based on well-explored principles [1, 8, 11, 12, 18, 19, 32–34]. Rather than repeating the standard details of such protocols, we describe how we adapt the approach to make it easy to add

BFT to a range of cluster applications. We begin with an overview, and the subsections that follow provide details.

As the figure illustrates, executing a client request involves three modules: request quorum, order, and execution. A UpRight client deposits its request at a *request quorum* (RQ), which stores the request, forwards a digest of the request to the order module, and supplies the full request to the execution module. The order module produces a totally ordered sequence of batches of request digests. The execution module embodies the application’s server, which executes requests from the ordered batches and produces replies.

This section discusses three aspects of UpRight agreement that depart from prior BFT protocols. First, the new RQ stage helps avoid complex corner cases and avoids sending large requests through the order stage. Second, the agreement protocol is new: it combines ideas from three prior systems (Zyzyva’s speculative execution [18], Aardvark’s techniques for robustness [11], and Yin et al.’s techniques for separating agreement and execution [34]), and its implementation is robust to widespread outages. Third, to minimize replication costs, the UpRight prototype allows one to separately configure  $u$ , the number of failures it can tolerate while remaining up, and  $r$  the number of failures it can tolerate while remaining right.

### 4.1 RQ: Request quorum

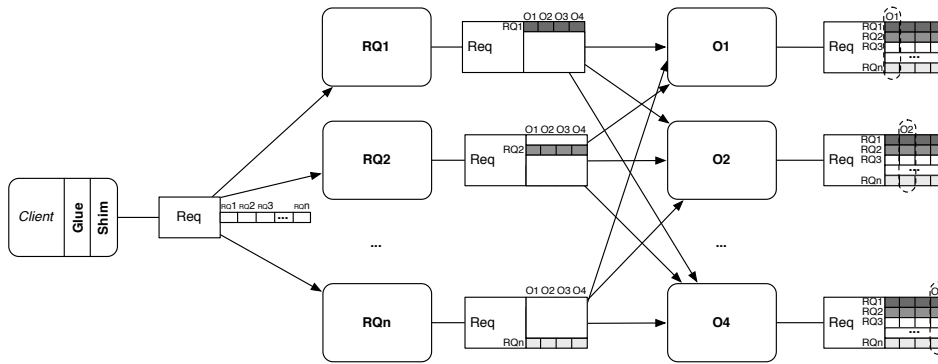
UpRight introduces the new request quorum (RQ) stage for two reasons: (1) to validate requests and thereby avoid expensive corner cases with inconsistent client MACs and (2) to separate the data path from the control path so that large requests are not sent through the order stage.

**Validating requests.** The use of MAC authenticators rather than digital signatures is a vital optimization to BFT replication systems [8]. If node  $A$  wants to send a message to  $n$  other nodes  $B_1 \dots B_n$ , it generates a MAC authenticator—an array of  $n$  message authentication codes (MACs)—by securely hashing the message with  $n$  distinct secret keys. Because secure hash generation is much faster than generating a public key signature, this optimization significantly reduces overheads.

Unfortunately, MAC authenticators do not provide non-repudiation, so if entries are inconsistent,  $B_1$  may validate the message as authentic but  $B_2$  may not. In many prior protocols, such inconsistent MACs force the system down complex alternate execution paths including timeouts, view changes, or signature generation, so a single faulty client can reduce throughput by orders of magnitude [11].

One solution is to have clients sign requests [11]. Given asymmetric public key signature schemes in which verifying a signature is cheap relative to generating one, this approach is affordable if client costs are ignored. Unfortunately, for the cluster services we target, the service provider is typically paying for both the servers and the clients, so ignoring client overheads is not appropriate.

Instead, we adapt matrix signatures [3] to allow us to retain most of the the performance benefits of MACs while avoiding the inconsistent-MAC corner cases. As Figure 4 illustrates, a client sends its request to the RQ nodes with a MAC authenticator. Each RQ node checks its entry in the authenticator, and if the request appears valid, the RQ node sends the request to the order nodes with a MAC authenticator for them. By transforming a single client request into



**Figure 4: The RQ stage generates matrix signatures to produce digests of client requests that all correct order nodes can validate.**

a quorum of RQ requests, we ensure that order nodes reach consistent conclusions.

**Request digest.** As Figure 3 indicates, RQ nodes send digests of large requests (more than 64 bytes in our prototype) to the order nodes rather than sending the full requests. However, if order nodes operate on digests rather than full requests, we must ensure that an execution node can expand a digest into a full request. An RQ node therefore stores each request before forwarding its digest. When an execution node receives an ordered digest, it notifies all RQ nodes that the request has been ordered and fetches the request from one of them.

RQ nodes bound the state to buffer requests with two limits. First, an RQ node will store at most one unordered request per client and ignore any new requests from that client until it learns that the previous request has been ordered. Second, as discussed below, Execution nodes produce a checkpoint every CP\_INTERVAL batches, and they never fetch requests older than the two most recent checkpoints.

## 4.2 Fast, robust ordering

UpRight’s order protocol takes from Zyzyva [18] a fast path based on speculative execution in which (1) a client sends its request (via the RQ) to the primary, (2) the primary accumulates a batch of requests, assigns the batch a sequence number, and sends the ordered batch of requests to the replicas, the replicas send the ordered batch of requests and a hash of the history of prior ordered batches to the execution nodes, and (3) the execution nodes accept and execute the batch of ordered request only if a sufficient number of ordering decisions and histories match.

UpRight’s order protocol takes from Aardvark [11] techniques for ensuring robust performance even when faults occur: consistent validation of client requests (via the RQ rather than signatures), resource scheduling and request filtering to prevent faulty nodes from consuming excess resources, and self-tuning performance requirements to replace a slow primary.

UpRight’s order protocol takes from Yin et al.’s protocol [34] separation of ordering and execution, which reduces the number of application replicas required.

Combining these ideas raises some new issues and provides some new benefits.

The main issue in combining Zyzyva’s speculation with Aardvark’s robustness is that Zyzyva relies on clients to trigger slow-path processing when responses from replicas don’t match, but Aardvark warns against allowing potentially faulty nodes to unilaterally drive the system to expensive execution paths. Therefore, instead of relying on clients to trigger full agreement among replicas as in Zyzyva, UpRight’s agreement protocol discovers and corrects divergence among replicas when the order nodes communicate to checkpoint their state. Because the protocol waits for order checkpoints to correct divergence, faulty order nodes can add latency. We take a checkpoint every CP\_INTERVAL batches, which must be low enough to ensure a tolerable latency for correcting divergence; we use CP\_INTERVAL=200 in our prototype. To reduce recovery latency, during periods of low offered load, the primary has the option of initiating the full agreement path. The self-tuning performance requirements ensure that if a faulty primary significantly slows progress by using the full agreement path when it is not helpful, it will be replaced.

Separating the order and execution phases fixes a significant limitation in Zyzyva by reducing the cost of correcting divergence. In Zyzyva, for example, a faulty primary can cause a correct replica to execute requests in the wrong sequence. The Zyzyva protocol ensures that clients will not act on the resulting wrong response, but it requires the divergent replica to correct itself by fetching a checkpoint and log of subsequent requests from the other replicas, apply the checkpoint, and replay the log. Since the checkpoint includes the state of the application and the last response sent to each client, this recovery from misspeculation can be expensive.

Conversely, in UpRight the order checkpoint is small because it contains only the identifier of the last request ordered by each client and a small amount of global state. In a system with  $c$  clients and 8 byte client and request identifiers, the order checkpoint is only  $16c + 58$  bytes. Also, log replay simply entails updating the last ordered table. Notice that correct UpRight Execution nodes act on quorums of ordered requests and never diverge from one another.

**Separating order and execution revisited.** The separation of the order and execution stages requires coordination between the two to coordinate garbage collection and to ensure agreement on execution checkpoint state.

After executing batch  $B$  with sequence number  $(i * CP\_INTERVAL)$ , an execution node begins to produce execution checkpoint  $i$ . Once that checkpoint has been produced, the execution node sends a hash of the checkpoint with a MAC authenticator to the order nodes. An order node refuses to order batch  $B'$  with sequence number  $(i + 1 * CP\_INTERVAL)$  until it has received a quorum of matching hashes for execution checkpoint  $i$ . Thus to order batch  $B'$ , order nodes implicitly agree that a quorum of execution nodes report identical checkpoints.

If an execution node falls behind (e.g., it misses some batches because of a network fault or a local crash-recover event), it catches up by sending the order nodes its current sequence number, and the order nodes respond with the hash and sequence number of the most recent execution checkpoint and with the ordered batches since that checkpoint. The execution node then fetches the execution checkpoint from its peers and processes the subsequent ordered batches to get to the current state.

This coordination is necessary for two reasons. First, it allows order nodes and execution nodes garbage collect their logs and checkpoints. Once batch  $B''$  with sequence number  $i + 2 * CP\_INTERVAL$  has been ordered, the order and execution nodes garbage collect their logs and checkpoints up to and including batch  $B$ . Second, it is necessary for all correct execution nodes to eventually agree on the checkpoint that represents the state of the system after processing batch  $B$ . The execution nodes cannot reach agreement on their own, so they rely on the order nodes.

**Surviving widespread outages.** PBFT explicitly requires that all servers have uninterruptable power supplies and assumes that at most  $f$  nodes simultaneously fail so that a server's memory can be treated as stable storage [8]; many subsequent approaches have inherited these assumptions [1, 11, 12, 18, 19, 32–34]. If an environment violates these assumptions, safety could be compromised—nodes' states may no longer reflect the effects of previously executed requests or nodes' states may diverge from one another. In contrast, the crash fault tolerant lineage from Paxos typically relies on the file system for stable storage [6, 31]. Although there exist interesting techniques for safeguarding DRAM against failures [23], we judge that UpRight can be most easily adopted if it makes minimal assumptions about the environment in which it runs.

We therefore engineer UpRight to be safe and eventually live even if all nodes crash and lose the contents of their memory and eventually all but  $u$  nodes in each stage recover and resume operation. To that end, an RQ node logs requests before sending digests to be ordered, and an order node logs ordering decisions before sending them to be executed. Execution nodes create stable checkpoints every  $CP\_INTERVAL$  batches so that RQ and order nodes can garbage collect their state. These extra' disk writes (compared to the PBFT lineage) are to sequential logs and are pipelined with other batches of requests to limit the impact on throughput.

### 4.3 Replication cost

UpRight allows applications to add Byzantine fault tolerance to their existing crash tolerance at minimal cost. Since users may not want to pay to tolerate as many Byzantine failures as fail-stop failures, UpRight separately configures

$u$ , the total number of failures it can tolerate and remain live and  $r$  the number of those failures that can be commission failures while maintaining safety [13, 21]. UpRight also separates its order and execution stages [34] to allow users to minimize the number of application replicas.

As is standard and assuming  $u \geq r$ , UpRight requires  $u + r + 1$  application replicas for the execution stage and  $2u + r + 1$  nodes for the order stage. The RQ stage requires  $2u + r + 1$  nodes to ensure that the order stage orders every properly submitted request and that the execution stage can fetch any request that is ordered:  $2u + r + 1$  RQ nodes ensures that the primary order node can always be sent a request with  $u + r + 1$  valid entries in the matrix signature, which assures the primary that all order nodes will see  $u + 1$  valid entries, which assures the order nodes that execution nodes can fetch the full request even if  $u$  RQ nodes subsequently fail.<sup>2</sup> Logical nodes can be multiplexed on the same physical nodes, so a total of  $2u + r + 1$  physical nodes can suffice.

The comparison with a crash fault tolerant system is thus simple: if one has a CFT system that remains *up* despite  $u$  omission failures, then to use UpRight to also remain *right* despite  $r$  commission failures, add  $r$  machines. For example, a CFT system that tolerates 2 failures ( $u = 2$ ) would require 5 machines, all of which might run some Paxos [20, 24] variant and three of which also include replicas of the application server. In comparison, an UpRight BFT system that tolerates 2 failures of which 1 may be a commission failure ( $u = 2, r = 1$ ) would require 6 machines, all of which run RQ and order nodes and 4 of which also include replicas of the application server.

Simply counting nodes presents an incomplete story about system costs because different nodes have different resource demands and hence different costs. For example, in many systems, execution nodes will have more work to do than order or RQ nodes. Rather than counting nodes, our experiments measure the resources consumed by each instance of each stage to capture the fundamental costs of UpRight.

**Preferred quorums and hot spares.** In the worst case UpRight requires as many as  $2u + r + 1$  RQ nodes,  $2u + r + 1$  order nodes, and  $u + r + 1$  execution nodes, but during synchronous periods when no faults are observed it is safe, live, and able to use its optimized fast path with  $u + r + 1, 2u + r + 1,$  and  $u + 1$  RQ, order, and execution nodes, respectively. An UpRight deployment can reduce its processing overheads and average replication cost by using preferred quorums or hot spares.

Using *preferred quorums* means optimistically sending messages to the minimum number of nodes and resending to more nodes if observed progress is slow.

Using *hot spares* means delaying allocation of some nodes until needed and then allocating those nodes from a pool of hot spares [33]. Supporting such configurations is easy—UpRight, like its predecessors, is configured to be safe and eventually live in an asynchronous system that will eventually have a sufficiently long synchronous interval, so off-line

<sup>2</sup>Note that if we also want to ensure that an order node can quickly force a view change when it suspects that a faulty primary is failing to order a request, we would require  $2u + 2r + 1$  RQ nodes; UpRight avoids the extra  $r$  RQ nodes while still ensuring that all requests eventually are executed by instead relying on the frequent, periodic rotation of the primary discussed in Section 4.2.

hot spares can simply be regarded as unusually slow nodes when activated.

In our prototype, we use preferred quorums to minimize RQ overheads. A client first transmits its request to  $u + r + 1$  RQ nodes. After a timeout, the client resends to all of the RQ nodes. The first time an RQ node receives a client request it forwards the request only to the primary; upon any subsequent receipt of that request the RQ node forwards the request to all order nodes. The RQ nodes are also good candidates for on-demand allocation via hot spares because activating a RQ node does not require transferring state from other nodes—the RQ module starts in a clean state and can immediately begin processing client requests.

The order nodes’ agreement algorithm benefits from fast path speculative agreement when all nodes are correct and present [18], so use of preferred quorums or configuring an order node as a hot spare may hurt performance.

Using preferred quorums to send requests to subsets of execution nodes or making an execution node a hot spare may be most attractive if the application has only a small amount of state that must be transferred before a hot spare begins processing; otherwise it may be more challenging to configure execution nodes as hot spares without risking unacceptably long pauses when a spare is activated. ZZ [33] masks some of this start-up time by activating an execution node once the checkpoint for its in-memory data structures has been transferred and then transferring its on-disk state in the background, but we have not implemented this optimization in the UpRight prototype.

#### 4.4 Implementation and performance

We implement the agreement protocol in Java and regretably must name the prototype JS-Zyzyvark (Java Stable Zyzyva + Aardvark); J-Zyzyvark refers to the configuration where we omit writing to disk for comparison with prior Byzantine agreement protocols and to expose other bottlenecks in the protocol. We believe that a Java-based solution is more palatable for widespread deployment with the Java-based Zookeeper and HDFS systems than a C implementation despite the performance hit we take. We also note that logging actions to disk places a ceiling on throughput so the benefits of further optimization may be limited.

We run our servers on 3GHz dual-core Pentium-IV machines, each running Linux 2.6 and Sun’s Java 1.6 JVM. We use the FlexiProvider [14] cryptographic libraries for MACs and digital signatures. Nodes have 2GB of memory and are connected via a 100Mbit/s Ethernet. Except where noted, we use separate machines for RQ, order, and execution.

UpRight’s core library (shims, RQ, order, execution) comprise 20,403 lines of code (LOC), and the hybrid checkpoint/delta glue libraries comprise 1602 LOC.

**Response time and throughput.** Figure 5 shows the throughput and response time of J-Zyzyvark and JS-Zyzyvark. We vary the number of clients issuing 1 byte or 1 KB null requests that produce 1 byte or 1 KB responses and drive the system to saturation. We configure the system to tolerate 1 fault ( $u = r = 1$ ).

For small requests J-Zyzyvark’s and JS-Zyzyvark’s peak throughputs are a respectable 5.5 and 5.1 Kops/second, which suffices for our applications. They are comparable to unmodified Zookeeper’s peak throughput for small read/write requests, and they appear sufficient to support an HDFS

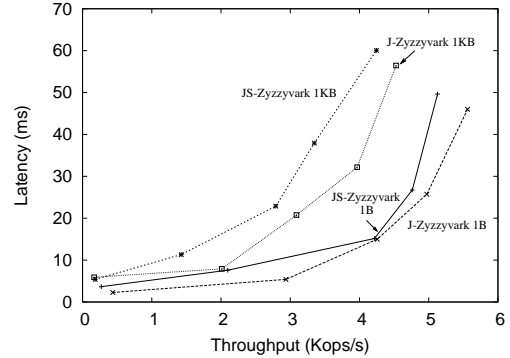


Figure 5: Latency v. throughput for J-Zyzyvark and JS-Zyzyvark.

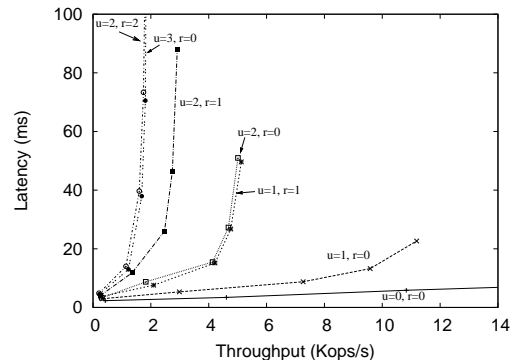


Figure 6: Latency v. throughput for JS-Zyzyvark configured for various values of  $r$  and  $u$ .

installation with a few thousand active clients and Data-Nodes. Peak throughputs fall to 4.5 and 4.2 Kops/second for a workload with larger 1KB requests and 1KB replies.

For comparison, on the same hardware Clement et al. [11] measure small request throughputs of 7.6, 23.8, 38.6, 61.7, and 66.0 Kops/s for the C/C++-based HQ [12], Q/U [1], Aardvark [11], PBFT [8], and Zyzyva [18]. For environments where performance is more important than portability or easy packaging with existing Java code bases, we believe a well-tuned C implementation of Zyzyvark with writes to stable storage omitted would have throughput between that of Aardvark and Zyzyva—our request validation and agreement protocols are cheaper than Aardvark’s, but our request validation is more expensive than Zyzyva’s.

**Other configurations.** Figure 6 shows small-request performance as we vary  $u$  and  $r$ . Recall that Zyzyvark requires  $2u + r + 1$  RQ and order nodes and  $u + r + 1$  execution nodes to ensure that it can tolerate  $u$  failures and remain up and  $r$  failures and remain right. Peak throughput is 11.1 Kops/second when JS-Zyzyvark is configured with  $u = 1$  and  $r = 0$  to tolerate a single omission failure (e.g., one

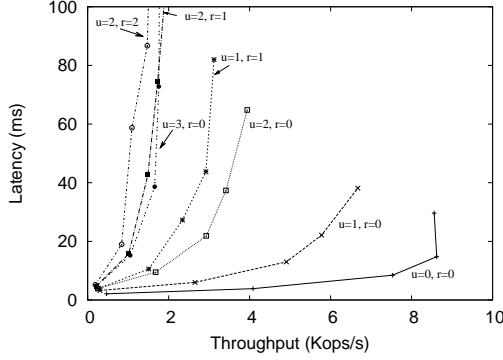


Figure 7: Latency v. throughput for JS-Zyzyvark configured for various values of  $r$  and  $u$  with RQ, Order, and Execution nodes colocated.

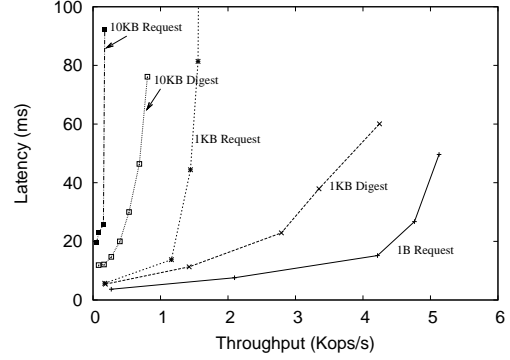


Figure 9: JS-Zyzyvark performance for 1B, 1KB, and 10KB requests, and for 1KB and 10KB requests where full requests, rather than digests, are routed through Order nodes.

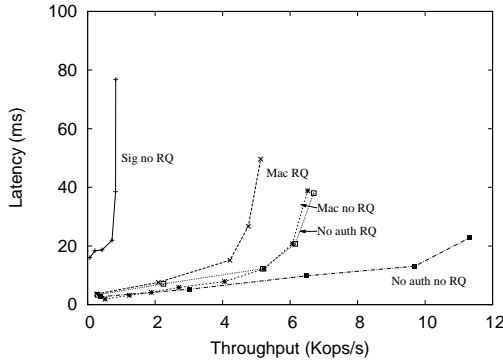


Figure 8: JS-Zyzyvark performance when using the RQ node and matrix signatures, standard signatures, and MAC authenticators. (1B requests)

crashed node), and throughput falls as the number of faults tolerated increases. For reference, we include the  $u = 0$   $r = 0$  line for which the system has just one RQ, order, and execution node and cannot tolerate any faults; peak throughput exceeds 22 Kops/s, at which point we are limited by the load that our clients can generate.

Figure 7 shows small request performance when the RQ, order, and execution nodes are co-located on  $2u + r + 1$  total machines. Splitting phases across machines improves peak throughput by factors from 1.67 to 1.04 over such co-location when any fault tolerance is enabled, with the difference falling as the degree of fault tolerance increases.

**Request authentication.** Figure 8 examines the throughput of the JS-Zyzyvark prototype configured for  $u = 1$   $r = 1$  and using different strategies for client request authentication. The *MAC RQ* line shows performance of the default JS-Zyzyvark configuration that relies on MAC-based matrix signatures formed at the RQ. In contrast, the *SIG no RQ* line omits the RQ stage and shows the significant perfor-

mance penalty imposed by relying on traditional digital signatures for request authentication as in Aardvark. The *MAC no RQ* line shows the performance that is possible in a system that relies on MAC authenticators and no RQ stage for client authentication as in PBFT. In a system where the robustness risk and corner-case complexity of relying on MAC authenticators as opposed to matrix signatures are viewed as acceptable, this configuration may be attractive. For comparison, the *no auth RQ* line shows performance when we use the RQ stage but turn off calculation and verification of MACs, and the *no auth no RQ* line shows performance when we eliminate the RQ stage and also turn off calculation and verification of MACs.

**Request digests.** Figure 9 tests the value of storing requests at the RQ so that the order stage can operate on digests rather than full requests. We configure the system for  $u = 1$   $r = 1$ . For small requests (under 64 bytes in our prototype), RQ sends full requests and the order nodes operate on full requests; the figure’s *1B Request* line shows performance for 1 byte requests. The *1KB Digest* and *10KB Digest* lines show performance for 1KB and 10KB requests when RQ nodes store requests and send request digests for ordering, and the *1KB Request* and *10KB Request* lines show performance with the request storage and digests turned off so that order nodes operate on full requests. Storing requests at the RQ more than doubles peak throughput for 1KB and 10KB requests.

## 5. ZOOKEEPER CASE STUDY

Zookeeper [35] is an open-source coordination service that, in the spirit of Chubby [6], provides services like consensus, group management, leader election, presence protocols, and consistent storage for small files.

Zookeeper guards against omission failures. However, because data centers typically run a single instance of a coordination service on which many cluster services depend [9], and because even a small control error can have dramatic effects [30], investing modest additional resources to protect the service against a wider range of faults may be attractive.

## 5.1 Baseline system

A Zookeeper deployment comprises  $2u + 1$  servers; a common configuration is 5 servers for  $u = 2$   $r = 0$ . Servers maintain a set of hierarchically named objects in memory. Writes are serialized via a Paxos-like protocol, and reads are optimized to avoid consensus where possible [8]. A client can set a *watch* on an object so that it is notified if the object changes unless the connection from the client to a server breaks, in which case the client is notified that the connection broke.

For crash tolerance, each server synchronously logs updates to stable storage. Servers periodically produce *fuzzy snapshots* to checkpoint their state: a thread walks the server’s data structures and writes them to disk, but requests concurrent with snapshot production may alter these data structures as the snapshot is produced. If a Zookeeper server starts producing a snapshot after request  $s_{start}$  and finishes producing it after request  $s_{end}$ , the fuzzy snapshot representing the system’s state after request  $s_{end}$  comprises the data structures written to disk plus the log of updates from  $s_{start}$  to  $s_{end}$ .

## 5.2 UpRight-Zookeeper

UpRight-Zookeeper is based on Zookeeper version 3.0.1. Given the UpRight framework, adding Byzantine fault tolerance to Zookeeper to produce UpRight-Zookeeper is straightforward. Our shims use standard techniques to add authenticators to messages and to send/receive them to/from the right quorums of nodes. We use the techniques described above to support watches via server push, to make time-based events happen deterministically across replicas at the same virtual time, and to canonicalize read-only replies. Zookeeper’s fuzzy snapshots align well with our hybrid checkpoint/delta approach; we modify Zookeeper to make the snapshots deterministic and identical across replicas using the copy on write approach.

The original Zookeeper server comprises 13589 lines of code (LOC). We add or modify 604 lines to integrate it with UpRight. The bulk of these changes involved modifying the checkpoint generation code to include all required state and integrate a helper process for use with the hybrid checkpoint/delta approach (347 LOC), glue code to handle communication between Zookeeper and our libraries (129 LOC), and making references to time and randomness deterministic across replicas (66 LOC). We also deactivate or delete some existing code. In particular, we delete 342 LOC that deal with asynchronous IO and multithreading, and we no longer use 5644 LOC that handle Zookeeper’s original replication protocols. We modify an additional 554 LOC to provide support for copy on write checkpointing.

## 5.3 Evaluation

We evaluate Zookeeper 3.0.1 and UpRight-Zookeeper running on the hardware described in Section 4.4. For Zookeeper, we run with the default 5 servers ( $u = 2$   $r = 0$ ). We then configure UpRight-Zookeeper to tolerate as many or more faults. In particular, we examine UpRight-Zookeeper with  $u = 2$   $r = 1$  for all phases to minimize the replication cost of adding commission failure tolerance while retaining at least Zookeeper’s original omission failure tolerance. We also examine a configuration that we refer to as  $u = 2+$   $r = 1$  that has  $u = 2$   $r = 1$  for the RQ and order stages and  $u_{exec} = 3$   $r_{exec} = 1$  for the execution stage; this configuration retains

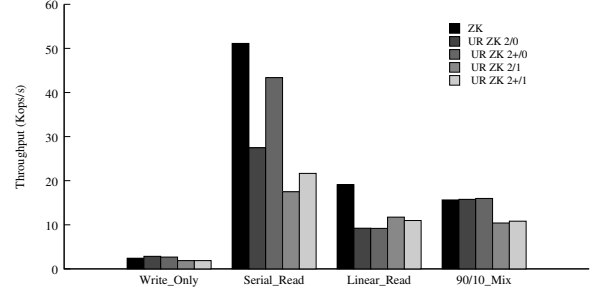


Figure 10: Throughput for UpRight-Zookeeper and Zookeeper for workloads comprising different mixes of 1KB reads and writes.

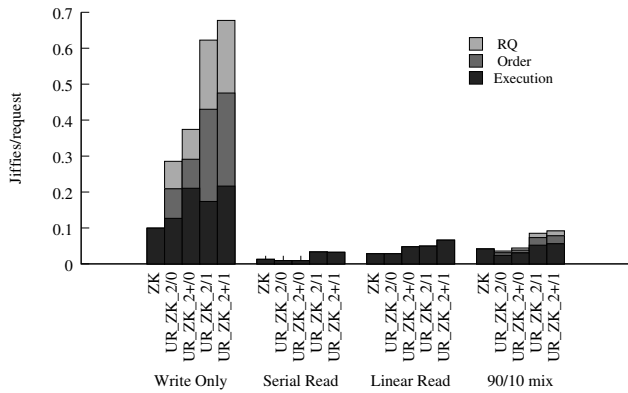
Zookeeper’s default 5 execution replicas. The results presented here rely on the helper process approach for checkpointing. We observe similar performance when using copy on write techniques.

In addition, we evaluate UpRight-Zookeeper’s performance in CFT configurations ( $r = 0$ ) to explore whether UpRight would be a suitable for new applications that want to support both CFT and BFT configurations using a single library. We evaluate the performance of UpRight-Zookeeper with  $u = 2$   $r = 0$  to match Zookeeper’s omission tolerance with the minimum degree of replication. We also evaluate a configuration that we refer to as  $u = 2+$   $r = 0$  that has  $u = 2$   $r = 0$  for the RQ and order stages and  $u_{exec} = 4$   $r_{exec} = 0$  for the execution stage; this configuration retains Zookeeper’s default 5 execution replicas.

Figure 10 shows throughput for different mixes of 1KB reads and writes.

For writes, the systems sustain several thousand requests per second. Nearly a decade of effort to improve various aspects of BFT agreement [1, 8, 11, 12, 18, 19, 32–34] have paid off: when  $r = 1$ , UpRight-Zookeeper’s write throughput is 77% of Zookeeper’s for both  $u = 2$  and  $u = 2+$ . UpRight also appears to provide competitive write performance for CFT configurations: for  $u = 2$  or  $u = 2+$  and  $r = 0$  UpRight-Zookeeper’s throughput with  $r = 0$  and either  $u = 2$  or  $u = 2+$  is more than 111% of Zookeeper’s.

For reads that can accept serializability for their consistency semantics, both Zookeeper and UpRight-Zookeeper exploit the read-only optimization to skip agreement and issue requests to a quorum of  $r + 1$  execution nodes that have processed the reader’s most recent write. Both systems’ read throughputs are many times their write throughputs, but in configurations where Zookeeper queries fewer execution nodes or has more total execution nodes, its peak throughput can be proportionally higher. For example, when Zookeeper sends read requests to 1 server and spreads these requests across 5 execution replicas, we expect to see about 2.5 times the throughput of a configuration where UpRight-Zookeeper sends read requests to 2 servers (for  $r = 1$ ) and spreads them across 4 execution replicas. When UpRight-Zookeeper is configured to tolerate commission failures, it pays additional CPU overheads for cryptographic checksums but saves some network overheads by having only one execution node send a full response and having the others send a hash [8]. Overall, UpRight-Zookeeper’s serializable read throughput ranges from 17.5 Kops/s to 43.4 Kops/s, which is 34% to 85% of Zookeeper’s 51.1 Kops/s throughput.



**Figure 11: Per-request CPU consumption for UpRight-Zookeeper and Zookeeper for a write-only workload. The  $y$  axis is in jiffies. In our system, one jiffy is 4 ms of CPU consumption.**

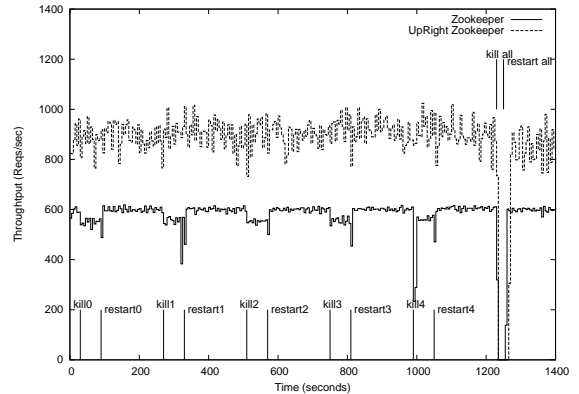
Although reading identical results from a properly chosen quorum of  $r + 1$  servers can guarantee that the read can be sequenced in a global total order, the position in the sequence may not be consistent with real time: a read by one client may not reflect the most recently completed write by another. So, some applications may opt for the stronger semantics of linearizability. For linearizable reads, UpRight-Zookeeper can still use the read only optimization, but it must increase the read quorum size to  $n_{exec} - r_{exec}$ . To enforce linearizability the original Zookeeper issues a *sync* request through the agreement protocol and then issues a read to the same server, which ensures that server has seen all updates that completed before the sync.

The last group of bars examines performance for a mix of 90% serializable reads and 10% writes. When UpRight-Zookeeper is configured to tolerate  $r = 1$  commission failures, its performance is over 66% of Zookeeper’s. When it is configured to tolerate omission failures only, its performance is comparable to Zookeeper’s.

Although the throughputs of our BFT configurations are comparable to those of the original CFT system, the extra guarantees come at a cost of resource consumption. Figure 11 shows that each request consumes significantly more CPU cycles under UpRight-Zookeeper than under Zookeeper. The graph shows per-request CPU consumption when both systems are heavily loaded; we observe similar results across a wide range of loads.

We note that although using Java rather than C for agreement only modestly hurts our throughput for this application, it does significantly increase our resource consumption. Judging by peak throughputs on similar hardware, agreement protocols like PBFT and Zyzzyva may consume an order of magnitude fewer CPU cycles per request than J-Zyzzyvark. Future work is needed to see if a C realization of UpRight’s agreement protocol would provide a lower cost option for deployments willing to shift from Java to C.

Figure 12 shows how throughput varies over time as nodes crash and recover. For this experiment we compare against Zookeeper 3.1.1 because it fixes a bug in version 3.0.1’s log garbage collection that prevents this experiment from completing. The workload is a series of 1KB writes generated by 16 clients, and we compare Zookeeper ( $u = 2$   $r = 0$ ) with



**Figure 12: Performance v. time as machines crash and recover for Zookeeper and UpRight-Zookeeper.**

UpRight-Zookeeper configured with  $u = 2 + r = 1$ . At times 30, 270, 510, 750, and 990 we kill a single execution node and restart it 60 seconds later. At time 1230 we kill all execution nodes and restart them 20 seconds later. Both systems successfully mask partial failures and recover quickly after a system-wide crash-recover event.

## 6. HDFS CASE STUDY

The Hadoop Distributed File System (HDFS) [16] is an open-source cluster file system modeled loosely on the Google File System [15]. It provides parallel, high-throughput access to large, write-once, read-mostly files.

UpRight-HDFS enhances HDFS by (1) eliminating a single point of failure and improving availability by supporting redundant NameNodes with automatic failover and (2) providing end-to-end Byzantine fault tolerance against faulty clients, DataNodes, and NameNodes.

### 6.1 Baseline system

An HDFS deployment comprises a single *NameNode* and many *DataNodes*. Files are broken into large (default 64MB) blocks, and by default each block is stored on three DataNodes. The NameNode keeps the file name to block ID mappings and caches the block ID to DataNodes mappings reported by DataNodes as soft state.

To write a new block, a client requests a new block ID from the NameNode, the NameNode selects a block ID and a list of DataNodes, the client sends a write comprising the block ID, the data, a list of 4-byte CRC32 checksums for each 512 bytes of data, and a list of DataNodes to the nearest listed DataNode, that DataNode stores the data and checksums, forwards the write to the next DataNode on the list, and reports the completed write to the NameNode. After the DataNodes acknowledge the write, the client sends a *write complete* request to the NameNode; the write complete request returns once the NameNode knows that the data has reached the required number of DataNodes. To read a block, a client requests a list of the block’s DataNodes from the NameNode, sends the read request to a DataNode, and gets the data and checksums in reply.

DataNodes send periodic heartbeats to the NameNode. After a number of missed heartbeats, the NameNode declares the DataNode dead and replicates the failed node's blocks from the remaining copies to other DataNodes.

The NameNode checkpoints its state to a file with the help of a Secondary NameNode. The NameNode writes all transactions to a series of log files. Periodically, the Secondary fetches the most recent log file and the current checkpoint file. The Secondary then loads the checkpoint, replays the log, and writes a new checkpoint file. Finally, the Secondary sends the new checkpoint file back to the NameNode, and the NameNode can reclaim the corresponding log file. If a NameNode crashes and recovers, it first loads the checkpoint and then replays the log.

The fault tolerance of the baseline HDFS system is a bit unusual. The checksums at the DataNodes protect against some but not all Byzantine failures. For example, if a DataNode suffers a fault that corrupts a disk block but not the corresponding checksum, then a client would detect the error and reject the data, but if a faulty DataNode returns the wrong block and also returns the checksum for that wrong block, a client would accept the wrong result as correct. In its default configuration, HDFS can ensure access to all data even if two DataNodes fail by omission, and it can ensure that it returns correct data for some but not all commission failures of up to two DataNodes. We will summarize HDFS DataNodes' fault tolerance as  $u = 2$   $r = 0/2$ .

HDFS's Secondary NameNode's role is just to compact the log into the checkpoint file, and there is no provision for automatically transferring control from the NameNode to the Secondary NameNode. If the NameNode suffers a catastrophic failure, one could imagine manually reconfiguring the system to run the NameNode on what had been the Secondary's hardware, but recent updates could be lost. An HDFS NameNode's fault tolerance is  $u = 0$   $r = 0$ .

## 6.2 UpRight-HDFS

Given the UpRight framework, adding Byzantine fault tolerance to HDFS is straightforward.

### 6.2.1 UpRight-NameNode

Adapting the HDFS NameNode to work with UpRight requires modifications to less than 1750 lines of code. The bulk of these changes, almost 1600 lines, relate to checkpoint management and generation. In particular, we add about 730 lines to include additional state in checkpoints. For example, we include mappings from block IDs to DataNodes in a NameNode's checkpoints—although we still treat these mappings as soft state that expires when a DataNode is silent for too long, including this state in the checkpoint ensures that NameNode replicas processing a request agree on whether the state has expired or not. In addition, we add about 830 lines to modify the logs to record every operation that modifies any NameNode state rather than only the modifications to the file ID to block ID mapping.

The other major change needed to make the HDFS NameNode compatible with UpRight is removing sources of nondeterminism from its request execution path. These changes affect under 150 lines and fall into 3 categories. We replace 5 references to local system time with references to the time provided by the order nodes for the current batch of request. Similarly, we modify 20 calls to *random()* so that they are all seeded by the agreed upon order time. The final step to

removing nondeterminism is disabling the threads responsible for running a variety of periodic background jobs and instead executing those tasks based on the logical time specified by the order nodes.

### 6.2.2 UpRight-DataNode

We originally imagined that we would replicate each DataNode as a BFT state machine and reduce the application-level data replication in light of the redundancy in the BFT DataNode “supernodes.” Although academically pure, simply using a black box state machine replication library to construct BFT data nodes would have changed the replication policies of the system in significant and perhaps undesirable ways. For example, HDFS's default data placement policy is to store the first copy on a node in the same rack as the writer, the second copy on a node in another rack, the third copy on a different node in the same rack as the second, and additional copies on randomly selected, distinct nodes. Further, if a DataNode fails and is replaced, HDFS ends up spreading the recovery cost approximately evenly across the remaining DataNodes. Additionally, if a new DataNode is added, the system gradually makes use of it. Although one could imagine approximating some of these policies within a state machine replication approach, we instead leave the (presumably) carefully-considered HDFS DataNode replication policies in place.

To that end, our UpRight-DataNode makes a few simple changes to the existing DataNode. The main changes are to (1) add a cryptographic subblock hash on each 64KB subblock of each 64MB (by default) block and a cryptographic block hash across all of a block's subblock hashes and (2) store each block hash at the NameNode. In particular, DataNodes compute and store subblock and block hashes on the writes they receive, and they report these block hashes to the NameNode when they complete the writes. A client includes the block hash in its write complete request to the NameNode, and the NameNode commits a write only if the client and a sufficient number of DataNodes report the same block hash. As in the existing code, clients retry on timeout, the NameNode eventually aborts writes that fail to complete, and the NameNode eventually garbage collects DataNode blocks that are not included in a committed write.

To read a block, a client fetches the block hash and list of DataNodes from the NameNode, fetches the subblock hashes from a DataNode, checks the subblock hashes against the block hash, fetches subblocks from a DataNode, and finally checks the subblocks against the subblock hashes; the client retries using a different DataNode if there is an error.

These changes require us to change or add 189 LOC at the client, 519 lines at the DataNode, and 238 lines at the NameNode.

Finally, we add the expected MACs and MAC authenticators to all messages with the exception of subblock hash and subblock data read replies from DataNodes to clients, which are directly or indirectly checked against the block hash from the NameNode.

## 6.3 Evaluation

In this section we compare UpRight-HDFS with the original. All experiments run on subsets of 107 Amazon EC2 *small* instances [4]. In each experiment, we have 50 DataNodes and 50 clients, and each client reads or writes a series of 1GB files. For both systems, we replicate each block to

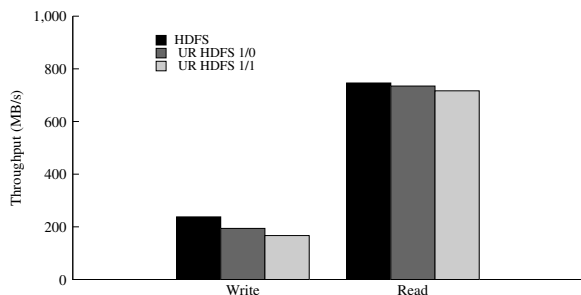


Figure 13: Throughput for HDFS and UpRight-HDFS.

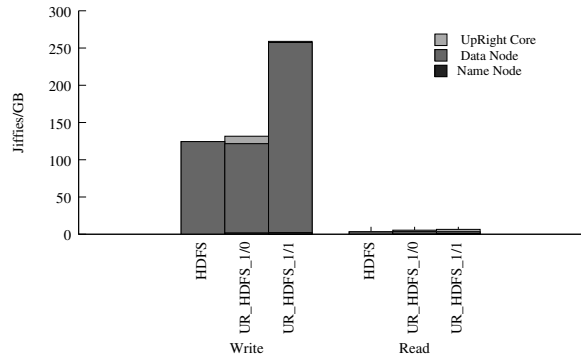


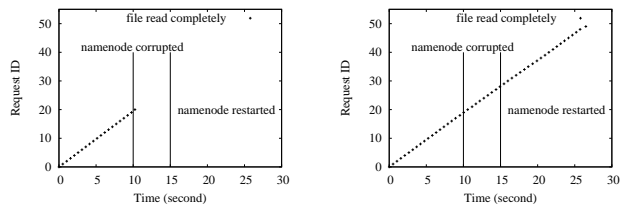
Figure 14: CPU consumption (jiffies per GB of data read or written) for HDFS and UpRight-HDFS.

3 DataNodes, giving  $u = 2$ ,  $r = 2/0$  for HDFS and  $u = 2$ ,  $r = 2$  for UpRight. HDFS’s NameNode is a single point of failure ( $u = r = 0$ ). For the UpRight-HDFS runs, we configure the NameNodes for  $u = r = 1$  and co-locate the RQ and order nodes. To evaluate UpRight’s ability to support CFT configurations, we also look at a  $u = 1$ ,  $r = 0$  configuration.

Figure 13 shows the throughput achieved with 50 clients and DataNodes. For both systems, write throughput is lower than read throughput because each block is written to three disks but read from one. Even with  $r = 1$ , UpRight-HDFS’s read performance is approximately equal to that of HDFS’s because only one DataNode is required to read and send the data. With  $r = 1$ , UpRight-HDFS’s write performance is over 70% of HDFS’s; the slowdown on writes appears to be due to added agreement for the replicated NameNode and the overheads of MAC computations for the DataNodes. With  $r = 0$ , the MAC computations are omitted and write performance is over 80% of HDFS’s; the compensation for this slowdown is the ability to remain available even if a NameNode crashes.

Figure 14 shows the CPU consumption for these workloads. When  $r = 1$ , UpRight-HDFS’s CPU costs are within a factor of 2.5 of the original for writes and within a factor of two for reads. Note that CPU consumption is one of the worst metrics for UpRight-HDFS; other system resources like the disks and networks have much lower overheads. When  $r = 0$ , the overheads are smaller—factors of 1.1 and 1.6 for writes and reads, respectively.

UpRight-HDFS incurs additional computational overheads for lower performance than HDFS. These costs come with a benefit as demonstrated by Figure 15. The two graphs plot



(a)

(b)

Figure 15: Completion time for requests issued by a single client. In (a), the HDFS NameNode fails and is unable to recover. In (b), a single UpRight-HDFS NameNode fails, and the system continues correctly.

completion time for requests issued by a single client that issues each request .5 seconds after the previous request completes. After 10 seconds of this workload we kill a NameNode and in the process corrupt its checkpoint log. We then restart the NameNode after an additional 5 seconds. Progress with the HDFS NameNode stops at 10 seconds when the log becomes corrupted. When the NameNode restarts 5 seconds later it immediately crashes again after attempting to load the corrupted log. In UpRight-HDFS, the absence of a single NameNode does not prevent progress. Additionally, when the failed NameNode restarts, it fetches a valid state from the other replicas and resumes correct operation rather than attempting to load its corrupted local log.

## 7. RELATED WORK

We stand on the shoulders of numerous recent efforts to make BFT a practical reality. PBFT [8] and its successors [1, 11, 12, 18, 19, 32–34] have significantly reduced the replication overhead and performance cost of agreement and state machine replication. Aardvark [11] and Prime [5] focus on providing good performance even when faults occur. UpRight most directly works to combine the high performance agreement of PBFT [8] and Zyzyva [18] with low-cost replication from separating order and execution [34] and with Aardvark’s robustness [11]. Notably, where Aardvark uses client signatures and Prime uses a 2 or 3 round pre-ordering stage that uses order node signatures to validate client requests, UpRight’s RQ stage ensures consistent validation of client requests using only MACs.

Commercial best practices for replication have evolved towards increasing tolerance to fail-stop faults as hardware costs fall, as replication techniques become better understood and easier to adopt, and as systems become larger, more complex, and more important. For example, once it was typical for storage systems to recover from media failures using off-line backups; then single-parity or mirrored RAID [10] became *de rigueur*; now, there appears to be increasingly routine use of doubly-redundant storage [15, 25, 29]. Similarly, although two-phase commit is often good enough—it can be always safe and rarely unlive—increasing numbers of deployments pay the extra cost to use Paxos [20, 24] three-phase commit [6, 31] to simplify their design or avoid corner cases requiring operator intervention [6].

Deployed systems increasingly include limited Byzantine fault tolerance aimed at high-risk subsystems. For example the ZFS [27], GFS [15], and HDFS [16] file systems provide

checksums for on-disk data [26]. As another example, after Amazon S3 was felled for several hours by a flipped bit, additional checksums on system state messages were added [30]. Although it may be cheaper to check for and correct faults at critical points than to do so end-to-end, we fear that it may be difficult to identify all significant vulnerabilities *a priori* and complex to solve them case by case with *ad hoc* techniques. A contribution of this paper is to explore cases when an end-to-end approach can be employed.

## 8. CONCLUSIONS

The purpose of the UpRight library is to make Byzantine fault tolerance (BFT) a viable addition to crash fault tolerance (CFT) for a range of cluster services.

If a designer has an existing CFT service, UpRight can provide an easy way to also tolerate Byzantine faults. We test UpRight by constructing BFT versions of Zookeeper and HDFS. Although our design choices in UpRight favor minimizing intrusiveness to existing applications over raw performance, our UpRight-Zookeeper and UpRight-HDFS implementations have performance comparable with the original systems, and they provide additional robustness against Byzantine clients and servers.

If a designer is building a new service, the UpRight library makes it straightforward to provide BFT, and it can be attractive even if the designer's first priority is CFT or if the designer is uncertain about the need for BFT: UpRight allows separate configuration of its tolerance of omission and commission failures, so it provides a simple crash fault tolerance library with competitive performance when it is configured for crash tolerance only ( $r = 0$ ). Compared to writing a crash fault tolerant replication protocol from scratch by, for example, implementing some Paxos variant, using a standard library like UpRight may be significantly simpler. Compared to using an existing CFT replication library, UpRight provides the added option of activating Byzantine fault tolerance at some future date or for some deployments.

## Acknowledgments

This work was supported in part by NSF grants CSR-PDOS-0509338 and CSR-PDOS-0720649.

## 9. REFERENCES

- [1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.
- [2] T. Abdollah. LAX outage is blamed on 1 computer. *Los Angeles Times*, Aug. 2007.
- [3] A. S. Aiyer, L. Alvisi, R. A. Bazzi, and A. Clement. Matrix signatures: From MACs to digital signatures in distributed systems. In *DISC*, 2008.
- [4] Amazon elastic compute cloud. <http://aws.amazon.com/ec2/>, Mar. 2009.
- [5] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *DSN*, 2008.
- [6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.
- [7] M. Calore. Magnolia suffers major data loss, site taken offline. *Wired*, Jan. 2009.
- [8] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4), 2002.
- [9] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.
- [10] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2), 1994.
- [11] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.
- [12] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, 2006.
- [13] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous byzantine consensus. Technical Report EPFL/IC/200499, École Polytechnique Fédérale de Lausanne, 2005.
- [14] The FlexiProvider Group. the FlexiProvider Project. <http://www.flexiprovider.de>.
- [15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.
- [16] Hadoop. <http://hadoop.apache.org/core/>.
- [17] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.
- [18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP*, 2007.
- [19] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *DSN*, 2004.
- [20] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.
- [21] L. Lamport. Lower bounds for asynchronous consensus. In *FuDiCo*, June 2003.
- [22] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3), 1982.
- [23] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP*, 1997.
- [24] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC*, 1988.
- [25] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, 2007.
- [26] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *SOSP*, 2005.
- [27] A. Rich. ZFS, sun's cutting-edge file system. Technical report, Sun Microsystems, 2006.
- [28] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4), 1990.
- [29] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *FAST*, 2007.
- [30] A. S. Team. Amazon S3 availability event: July 20, 2008. <http://status.aws.amazon.com/s3-20080720.html>.
- [31] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *SOSP*, 1997.
- [32] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, 2007.
- [33] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical BFT using virtualization. Technical Report TR14-08, University of Massachusetts, 2008.
- [34] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, 2003.
- [35] Zookeeper. <http://hadoop.apache.org/zookeeper>.