Commodifying Replicated State Machines with OpenReplica

Deniz Altınbüken, Emin Gün Sirer
Computer Science Department, Cornell University
{deniz,egs}@cs.cornell.edu

Draft: Not for Redistribution

Abstract

This paper describes OpenReplica, an open service that provides replication and synchronization support for large-scale distributed systems. OpenReplica is designed to commodify Paxos replicated state machines by providing infrastructure for their construction, deployment and maintenance. OpenReplica is based on a novel Paxos replicated state machine implementation that employs an object-oriented approach in which the system actively creates and maintains live replicas for user-provided objects. Clients access these replicated objects transparently as if they are local objects. OpenReplica supports complex distributed synchronization constructs through a multi-return mechanism that enables the replicated objects to control the execution flow of their clients, in essence providing blocking and non-blocking method invocations that can be used to implement richer synchronization constructs. Further, it supports elasticity requirements of cloud deployments by enabling any number of servers to be replaced dynamically. A rack-aware placement manager places replicas on nodes that are unlikely to fail together. Experiments with the system show that the latencies associated with replication are comparable to ZooKeeper, and that the system scales well.

1 Introduction

Developing distributed systems is a difficult task, in part because distributed systems comprise components that can and do fail and in part because these distributed components often need to take coordinated action through failures. A typical distributed application maintains state that needs to be replicated and distributed, as well as actively executing threads of control whose behavior needs to be controlled. We term the former process replication and the latter synchronization; together they are known as *coordination*. The recent emergence of the cloud as a mainstream commercial deployment environment has amplified the need for *coordination services*, infrastructure software that provides a replication and synchronization framework for distributed applications.

Building coordination services is a difficult task. ZooKeeper [17] and Chubby [4] have recently emerged as the predominant coordination services for large-scale distributed systems. While these two systems differ in the underlying consensus protocol they employ, they both provide the same basic mechanism; namely, ordered updates to a replicated file, with optional callbacks on updates. This approach suffers from three shortcomings. First, a file-based API requires an application to convert its replicated state into a serialized form suitable for storing in a file. Consequently, the serialized form of data stored in the filesystem typically differs from the programmatic view of an object as seen by the developer. Bridging this disconnect requires either costly serialization and deserialization operations or a level of indirection, where the file serves as a membership service to convert between the two views. Second, these services require an application to express its communication and synchronization behavior using an upcall-based API. In essence, these coordination services provide a publish-subscribe system. Consequently, applications need to be rewritten to subscribe to the appropriate upcalls that match their synchronization needs. Further, because these upcall events separate control-flow from data-flow, event handlers typically have to perform expensive additional operations to reestablish the event context, such as reading from the replicated state to determine the type of modification. Finally and most importantly, configuring and maintaining the resulting distributed application is operationally challenging. For example, maintaining replica sets to prevent service degradation often requires manual intervention to spawn new replicas and changes to update configurations. Since the ZooKeeper atomic broadcast protocol does not support dynamic updates to replica sets, migrating services requires a system restart.

In this paper, we present a novel *object-oriented* self-configuring and self-maintaining coordination service for large-scale distributed systems, called OpenReplica. OpenReplica is a public web service that instantiates and maintains user-specific coordination instances easily. OpenReplica operates on user-provided objects that define state machines, which it transforms into fault tolerant replicated state machines (RSMs) [26, 43]. These objects are used to maintain replication and synchronization data and can be crafted to meet an application's coordination needs. The system maintains a set of live replicas that can provide instant failover and employs consensus to keep the replicas in synchrony as the state of the replicated objects change through method invocations. OpenReplica can deploy replicated state ma-

chines in a manner that reduces the likelihood of correlated failures and monitors the liveness of this deployment over time. Critically, users can interact with the replicated object through an automatically generated object proxy or the OpenReplica web interface. Both of these options provide an API that is identical to the original, non-fault-tolerant object[1]. Analogous to RPC [8], OpenReplica provides a programmatic view that simplifies much of the complexity of interacting with a replicated, fault-tolerant object implementation. Similar in spirit to OpenDHT [38], the goal of OpenReplica is to make the construction, deployment and maintenance of a fault-tolerant object accessible to non-expert programmers.

The OpenReplica approach differs from ZooKeeper and Chubby in significant ways that stem from the difference between active object replication and passive file-based replication. Because OpenReplica provides transparent object replication, applications need not be revamped to use a file-based API. Clients need not even be aware that certain components have been replicated, which greatly simplifies the programming effort. Further, it eliminates the necessity of serializing and deserializing the object on the performance-critical path; in fact, the replicated object need not be serializable, as might be the case with state machines with actively executing background threads in them. Whereas a file-based API requires clients to retrieve the state from the coordination service, recover the state machine, advance it through one or more transitions and store it back in the coordination service. A client node that uses OpenReplica need not instantiate the object locally. Moreover, in coordination services based on wait-free primitives, such as ZooKeeper, write operations might fail under contention. The possibility of such a failure, and the corresponding need to redo the requested operation, requires object methods to either be idempotent or to be protected with additional, heavy-weight, blocking synchronization constructs. In contrast, because OpenReplica replicas apply operations according to a strict total order, there is no possibility of a write failure due to an out-of-date version mismatch. o This implies, in turn, that object methods can be written in a straightforward procedural manner; they need not be idempotent, resort to heavy-weight locks, or employ a transactional style. Finally, OpenReplica provides blocking client method invocations for enabling clients to easily implement complex synchronization constructs using existing and well-understood synchronization primitives, such as locks and semaphores. The system API obviates the need to map traditional synchronization code onto sequences of file operations and event upcalls.

[1]The modified, fault-tolerant interface differs solely in additional exceptions that pertain to a distributed implementation.

The OpenReplica implementation is based on the Paxos protocol [28, 29] for consensus and uses a novel combination of Paxos features to achieve higher performance and dynamicity. To maximize availability and provide flexibility in replica management, we support dynamic view changes where the acceptor and replica sets can be modified at run time. To achieve high performance, we employ a version of the Paxos protocol based on a light-weight implementation built on asynchronous events. And to provide integration with existing naming infrastructure and to enable clients to be directed to up-to-date replicas, OpenReplica implements name servers which can provide authoritative DNS name service as well as optional integration with Amazon's Route 53 [3].

OpenReplica has been used to build and deploy several fault-tolerant services, including a fault-tolerant logging service, a reliable group membership tracker, a reliable data store, a configuration service as well as reliable data structures, such as binary search trees, red black trees, queues, stacks, linked lists, and synchronization objects, such as distributed locks, semaphores, barriers and condition variables. These implementations show that OpenReplica enables programmers to build and deploy non-trivial fault-tolerant services simply by implementing a local object. The amount of engineering effort that went into these applications is substantially lower than specialized, monolithic systems built around Paxos agreement.

Overall, this paper makes three contributions. First, it presents the OpenReplica service for the construction, deployment and maintenance of Paxos-based replicated state machines (RSMs). Second, it describes the OpenReplica implementation for building practical Paxos RSMs which support high-throughput, dynamic view changes, fault-tolerance through rack-aware replica placement, client synchronization control through a multi-return primitive, and DNS integration. Finally, it compares the performance of OpenReplica to ZooKeeper, known for its high performance implementation, and demonstrates that the system achieves low latency during regular operation, quick recovery in response to failures, and high scalability in the size of the replicated state. Specifically, OpenReplica outperforms ZooKeeper by 15% on latency for 5 replicas and exhibits comparable failure recovery times.

The rest of this paper is structured as follows. Section 2 outlines the OpenReplica approach to replication for general-purpose objects. Section 3 describes the implementation of the system. Section 4 evaluates the performance of the system and provides a comparison to ZooKeeper. Section 5 places OpenReplica in the context of past work on coordination services and Section 6 summarizes our contributions.

2 Approach

OpenReplica is a public web service that instantiates and maintains application-specific coordination services. This section describes the design outline and rationale for the OpenReplica approach to providing coordination services in large-scale distributed systems.

The goals of OpenReplica are as follows:

- **Easy-to-use**: Defining, implementing, deploying and maintaining replicated state machines should be straightforward, even for non-expert programmers.
- **Transparent**: Replication and fault-tolerance techniques must not require disruptive changes to application logic. Rendering parts of an existing application fault-tolerant should not require extensive changes to the code base.
- **Dynamic**: It should be possible to change the location and number of replicas at run-time. The correct operation of the system should not depend on the liveness of statically designated clients.
- **High-Performance**: The resulting fault-tolerant system should exhibit performance that is comparable to state-of-the-art coordination services.

The underlying coordination infrastructure used by OpenReplica tackles these goals with an object-oriented approach centered around a *coordination object* abstraction. A coordination object consists of requisite data and associated methods operating on that data, which, together, define a state machine capable of stopping and restarting client executions. In essence, a coordination object specifies the application functionality to be made fault-tolerant, as well as defining the fault-tolerant synchronization mechanisms required to control the execution of an application. A user defines a coordination object as if it were a local Python object, hands it to OpenReplica, which then instantiates replicas on a set of servers and creates a distributed and fault-tolerant coordination object.

OpenReplica ensures that the coordination object replicas remain in synchrony by using the Paxos protocol to agree on the order of method invocations. There has been much work on employing the Paxos protocol to achieve fault-tolerance in specific settings [34, 36, 4, 32, 10, 20, 1], in which Paxos was monolithically integrated into a specific, static API offered by the system. In contrast, OpenReplica is a general-purpose, open service that enables any object to be made fault-tolerant. We illustrate the overall structure of a coordination object instance in Figure 1 and discuss each component in turn.

Binary Rewriting: OpenReplica uses binary rewriting to ensure *single-object semantics*; that is, users have the illusion of a single object both when specifying and invoking a coordination object. The implementation uses binary rewriting on the server side to generate a networked object suitable for replication from an object specification. This process involves the generation of server-side stubs and a control loop that translates local method invocations into Paxos consensus rounds. OpenReplica also uses binary rewriting to generate a client proxy object whose interface is identical to the original object. Underneath the covers, the client proxy translates method invocations into client requests, which comprise a unique client request id, method name, and arguments for invocation. The proxy marshals client requests and sends them to one of the replicas, discovered through a DNS lookup. Depending on the responses returned from the replica, the proxy is also capable of suspending and resuming the execution of the calling thread, thereby enabling a coordination object to control the execution of its callers.

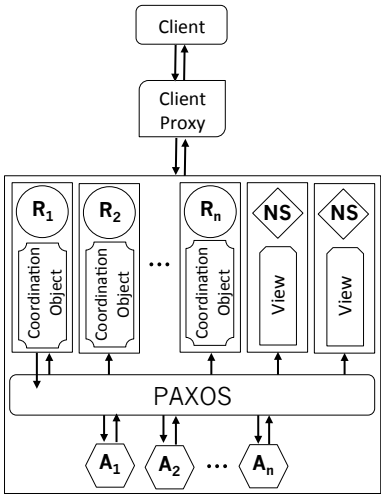


Figure 1: The structure of an OpenReplica coordination object. Clients interact with a coordination object transparently through the client proxy. The consistency and fault-tolerance of the user-defined coordination object is maintained by Replica (R), Acceptor (A) and Name Server (NS) nodes using the Paxos consensus protocol. Name server nodes keep track of the view of the system and reply to DNS queries for the coordination object with the latest view information.

RSM Synchrony: OpenReplica uses Paxos to ensure that the coordination object replicas are kept in synchrony. The central task of any RSM protocol is to ensure that all the replicas observe the same sequence of actions. OpenReplica retains this sequence in a data structure called *command history*. The command history consists of numbered *slots* containing client requests, corresponding to method invocations, along with their associated client request id, return value, and a valid bit indicating whether the operation has been executed. To ensure that operations are executed at most once, a replica checks the command history upon receiving a client re-

quest and, if the operation has already been executed, responds with the previously computed output. If the request has been assigned to a slot in the command history (i.e. a previous Paxos round has decided on a slot number for that request), but has not been executed yet, it records the client connection over which the output will be returned when the operation is ultimately executed. These two checks ensure that every method invocation will execute at most once, even in the presence of client retransmissions and failures of the previous replicas that the client may have contacted. If the client request does not appear in the command history, the receiving replica locates the earliest unassigned slot and proposes the operation for execution in that slot. This proposal takes place over a Paxos consensus round, which will either uncover that there was an overriding proposal for that slot suggested previously by a different replica (which will, in turn, defer the client request to a later slot in the command history and start the process again), or it will have its proposal accepted. These consensus rounds are independent and concurrent; failures of replicas may lead to unassigned slots, which get assigned by following rounds. Once a client request is assigned to a slot by a replica, that replica can propagate the assignment to other replicas and execute the operation locally as soon as all preceding slots have been decided. The replica then responds with the return value back to the client. Note that, while the propagation to other replicas occurs in the background, there is no danger of losing the agreed-upon slot number assignment, as the Paxos protocol implicitly stores this decision in a quorum of acceptor nodes at the time the proposal is accepted. For the same reason, OpenReplica does not require the object state to be written to disk. As long as there are less than a threshold f failures in the system, the state of the object will be preserved.

Dynamic Membership: Because coordination objects are fault-tolerant and long-lived, the system supports repositioning of the replicas dynamically during execution. Consequently, the connection between the client proxy and the replicas is established not by a static configuration file, but by a DNS lookup. DNS servers that participate in the agreement protocol track the membership of nodes in the replica set, and can thus respond to DNS queries with an up-to-date list of replicas. The names of the DNS servers in the parent-level DNS service is also updated whenever DNS servers come online, thus ensuring that the replica set can be located through the standard DNS resolvers. OpenReplica uses a fault-tolerant DNS service coordination object to keep track of the DNS servers for many coordination instances that are created for client coordination objects.

The end result of this organization is that the clients can treat the set of replicas as if they implement a single object. OpenReplica extends traditional Paxos RSM implementations with a novel multi-return mechanism to support two kinds of objects: synchronous and rendezvous objects.

```
class Account():
 def __init__(self, acctnumber, initbalance):
  self.number = acctnumber
  self.balance = initbalance

 def debit(self, amount):
  if amount >= self.balance:
    self.balance = self.balance - amount
    return True
  else:
    return False

 def deposit(self, amount):
  self.balance = self.balance + amount
  return True
```

Figure 2: Coordination objects do not include OpenReplica-specific code, they are implemented as if they are local objects.

Synchronous Objects

A *synchronous object* is a coordination object that encapsulates replicated state and provides associated methods to update this state, which do *not* change the execution state of their callers. Synchronous object methods execute to completion and return a result without suspending the caller.

Since synchronous objects are by far the most common type of object in distributed systems, OpenReplica makes it particularly easy to define and invoke them. Figure 2 shows a sample coordination object implementation for an online payment service. The Account class defines synchronous objects that hold a user's current account balance. The account has an identifier (an account number) and a balance, modified through `debit` and `deposit` methods. In effect, the object encloses the critical state that needs to be made fault-tolerant, and defines a state machine whose legal transitions are determined by the amount of money in the account. OpenReplica ensures that these operations are invoked in a consistent, totally-ordered manner.

What is noteworthy about this implementation is that it includes no replication-specific code. Neither the server-side object specification nor the user of the client proxy need be aware that the object is replicated and fault-tolerant. Single object semantics ensure that sound clients can be written in a straight-forward way, with only some additional exception handling for the cases where a partition or network failure results in a network timeout. In contrast, performing the same task with a file-based API in ZooKeeper or Chubby would require handling connections, serializing/deserializing persistent state, or perhaps using these systems to determine the membership of a set of live nodes which in turn manually implement an RSM.

The decision to maintain live instances and keep them in synchrony through agreement on the command history represents critical design tradeoffs. The advantage of agreeing on command history instead of object state is that it can support any object, even those that may contain active components, such as threads, and performs well even for large objects that may be too costly to serialize, such as large files. The downsides of this approach are two-fold: the command history can grow over time, a topic we address in the next section with garbage collection, and non-deterministic operations in methods may cause replica divergence if left unchecked, a topic we address in the next section through language mechanisms.

Rendezvous Objects

Making a distributed system fault tolerant typically requires synchronizing the activities of distributed components. OpenReplica accomplishes this with a novel *multi-return* primitive, supported for a class of objects dubbed *rendezvous objects*. Specifically, whereas synchronous objects support methods that simply execute to completion and return, rendezvous objects may block the calling client until further notice and resume it at a later point. Normally, clients of a replicated state machine perform synchronous method invocations, where every method invocation gets assigned to a slot in the replicated state machine history through consensus, and a result is returned to the client when the execution completes. In cases where the replicated state machine is used to synchronize clients, the method invocation may need to block the client and return as the result of a method invocation by another client. Note that this is not the same as the RSM itself blocking, though on casual observation, the two effects may seem the same. When the client is explicitly blocked, the RSM itself is free to take additional state transitions, prompted by operations issued by other clients. In contrast, when the RSM is blocked, it ceases to make progress and cannot uphold liveness requirements. OpenReplica avoids such blocking by enabling rendezvous objects to suspend and resume their calling clients.

The multiple return primitive greatly simplifies the implementation of objects used for synchronization. Figure 3 shows the implementation of a semaphore object in OpenReplica. The implementation follows a conventional semaphore implementation line-by-line. It keeps a count, a wait queue and an atomic lock and blocks or unblocks clients depending on the count value.

OpenReplica uses an extension over the underlying consensus protocol, where return values may be deferred. When a rendezvous object blocks its caller, a second bit in the command history is used to indicate that the calling client has been deferred. Later, any other command can cause previously deferred method calls to

```
class Semaphore():
 def __init__(self, count=1):
  self.count = int(count)
  self.queue = []
  self.atomic = Lock()

 def acquire(self, _concoord_cmd):
  with self.atomic:
   self.count -= 1
   if self.count < 0:
    self.queue.append(_concoord_cmd)
    raise BlockingReturn(None)
   else:
    return True

 def release(self, _concoord_cmd):
  with self.atomic:
   self.count += 1
   if len(self.queue) > 0:
    unblockcmd = self.queue.pop(0)
    unblocked = {}
    unblocked[unblockcmd] = True
    raise UnblockingReturn(None, unblocked)
```

Figure 3: Semaphore coordination object follows exactly from the traditional Semaphore implementation and it does not require instrumentation of upcall mechanisms to implement the expected behavior.

be resumed. Upon completion, these calls may yield actual returned values, which are returned to the client at a later time. The command history always records the time at which a call was deferred, as well as the later call that resumed the deferred method invocation. As a result, each method invocation in OpenReplica has associated with it not only its own results, but also the results of computations it resumed as a side-effect during its execution.

This enables a replica, replaying the object history, to make the same set of synchronization-related decisions as other replicas. On the client side, the intention of the RSM to block the client is communicated by an exception carried in the first response packet, which instructs the client proxy to block the calling thread on a local condition variable. A future, asynchronous response message for the same client request unblocks the thread and yields the result carried in the second response. Consequently, users can implement synchronization objects following conventional blocking constructs.

In contrast, systems with file-based APIs require esoteric, upcall-based implementations for synchronization control. For instance, a comparable barrier implementation is three times as long in ZooKeeper than its OpenReplica counterpart, requires intimate understanding of *znodes* and *watchers*, and has almost no code in common with textbook barrier implementations [17].

3 Implementation

Implementing a public, open web service for general purpose replication and coordination necessitates numerous design decisions on how to layer RSMs on top of the core Paxos protocol and how to maintain multiple instances of this distributed system. We present these implementation

details below, focusing on the design rationale.

3.1 Paxos

Paxos is used to achieve consensus among the replicas on the order in which client requests will be executed. By providing ordering guarantees, Paxos ensures that the replicated state machine behaves like a single remote state machine. Following the concise and lightweight multi-decree Paxos implementation described in [45], OpenReplica assigns a client-initialized request to a unique slot and communicates this assignment to the various nodes in the system.

OpenReplica employs two sets of nodes, replicas and acceptors. In OpenReplica, replicas keep a live copy of the replicated object, receive requests from clients, start a consensus round for each request and execute operations on the replicated state in the agreed upon order. Acceptors constitute the quorum keeping the consensus history, in effect providing memory for past decisions. Acceptors communicate solely with replicas and record the proposed client command and the highest Paxos ballot number they have seen for each slot. Consequently, replicas can use the acceptors to determine past history of proposals for each slot number, and to recover and resume past proposals in cases where they were only partially completed. At any time, a replica can retrieve the history of operations from acceptors to synchronize with other replicas. In the presence of special conditions like dynamic view changes, garbage collection, and nondeterministic inputs, the behaviors of replicas and acceptors are managed through additional mechanisms implemented upon the underlying RSM.

3.2 Meta commands

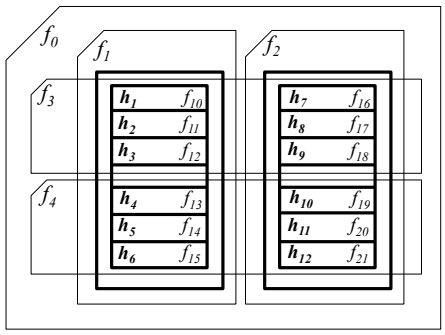
OpenReplica implements an internal control mechanism based on *meta commands* for managing replicas. Meta commands are special commands recognized by OpenReplica replicas that pertain to the configuration state of the replicated state machine as opposed to the state of the user-defined object. Meta commands are generated within OpenReplica and guaranteed to be executed at the same logical time and under the same configuration in every replica. This timing guarantee is required as the underlying protocol typically has many outstanding requests being handled simultaneously, and a change in the configuration would affect later operations that are being decided. For instance, a change in the set of Paxos acceptors would impact all ongoing consensus instances for all outstanding slots, and therefore needs to be performed in synchrony on all replicas.

To guarantee consistency through configuration changes, OpenReplica employs a *window* to define the number of non-executed operations a replica can have at any given time. To guarantee that meta commands are executed on the same configuration in every replica, the execution of a meta command is delayed by a window after it is assigned to a slot. For example, assume an OpenReplica setting where the window size is ω and the last operation executed by a replica is at slot α. Here, no other replica can initiate a consensus round for slots beyond $\alpha + \omega$. To initiate a consensus round for the next slot, a replica has to wait until after the execution of slot $\alpha + 1$. Therefore, when the command at $\alpha + \omega$ is executed, all replicas are guaranteed to have executed all meta commands through α. Hence, by delaying the execution of meta commands by ω, consistency of the Paxos related state can be maintained through a dynamic configuration change [31].

3.3 Dynamic Views

Long-lived servers are expected to survive countless network and node failures. To do so effectively, the system has to provide sufficient flexibility to move every component at runtime. OpenReplica achieves this by using meta commands to change the replica, acceptor and name server sets. Over time, an OpenReplica object may completely change the set of servers in its configuration, though adjacent configuration can modify at most f nodes because the state transfer mechanism used during view changes may temporarily keep new nodes from fully participating in the protocol. To maintain consensus history, new acceptors need to acquire past ballot number and command tuples from a majority of old acceptors for each past round. New acceptors transfer these ballots in the background until they have reached the current ballot. A meta command can then be issued to add the acceptor to the configuration, though there exists a window during which the acceptor may fall behind. Newly added acceptors ensure that they do not participate in the protocol until they have caught up by having heard from a majority of old acceptors for each past ballot. The acceptor addition mechanism suffers from a window of vulnerability during a configuration change where a newly added node consumes one of the f failure slots; past work has developed techniques for masking this window [32], though we have not yet implemented this technique due to its complexity. Replicas are easier to bring up, as any fresh replica will iterate through slot numbers, learn previously assigned commands by proposing NOOPs for each slot (whereupon the acceptors will notify the replica of previous assingments), and transition through states until it catches up. To speed this process up, the OpenReplica implementation allows a replica to fetch the command history from another replica en masse. The previous mechanism is then used to fill any gaps that could arise when the source replica is out of date.

Dynamic view changes in our system can be initiated externally, by a system administrator manually issuing



f_0: PDU Failure
$f_{1,2}$: Top-of-Rack Switch Failure
$f_{3,4}$: Cooling Unit Failure
f_{10-21}:Single Machine Failure

$h_1 : f_0\ f_1\ f_3\ f_{10}$
$h_4 : f_0\ f_1\ f_4\ f_{13}$
$h_7 : f_0\ f_2\ f_3\ f_{16}$
$h_{10} : f_0\ f_2\ f_4\ f_{19}$

Figure 4: Example of failure groups in a data center. Failure groups define set of nodes, whose failure depend strictly on the failure of one component. Data center outage defines f_0, failures of two top-of-rack switches define $f_{1,2}$, failures of two cooling units define $f_{3,4}$, and each machine failure defines $f_{10,21}$. OpenReplica uses the list of failure groups that a host is in, as an input to the greedy rack-aware replica placement algorithm.

commands, or internally, by a replica or the OpenReplica coordinator that detects a failure. In either case, the initiater typically brings up a nascent node, instructs it to acquire its state, and then submits a meta command to replace the suspected-dead node with the nascent one. To have the view change take effect quickly, independent of the rate of operations organically sent to the coordination object from clients, the initiater invokes ω NOOP operations.

3.4 Rack-Aware Replica Placement

The fault-tolerance of a distributed system is affected immensely when multiple servers fail simultaneously. These kinds of failures can happen if servers share crucial components such as power supplies, cooling units, switches and racks. Common points of failures define *failure groups* wherein a single failure would affect multiple servers. For instance Figure 4 illustrates possible failure groups in a data center by highlighting node sets that will be affected by the failure of a power distribution unit, a top-of-rack switch, a cooling unit and a machine. To prevent concurrent, non-independent failures, replica placement should be performed judiciously, minimizing the number of servers in the same failure group, and thus, subject to simultaneous failures due to the same root cause.

OpenReplica supports replica placement that takes failure scenarios into account, a feature that is sometimes called *rack-awareness*. During object instantiation, a user specifies the candidate set of hosts on which she can deploy replicas, along with a specification of their failure groups. Shown in Figure 4, a failure group specification is a free-form tuple that associates, with each server, the set of events that could lead to its failure. For instance, host h_7 shares common failure points f_0 and f_3 with h_1. OpenReplica places no limit on the number of failure groups, and is agnostic about the semantic meaning of each f_i. In this example, f_0 corresponds to a failure of a PDU that affects both racks shown in the figure, while f_3 corresponds to a cooling unit failure.

OpenReplica picks replicas using a greedy approach that achieves high fault tolerance. In particular, when picking a new host for a replica, acceptor or name server, it picks the host that maximizes the number of differences from the piecewise union of all existing hosts' failure groups. This greedy approach will not necessarily yield the minimally-sized replica group for tolerating a given level of failure, an open problem that has been tackled, in part, by other work [21]. Since OpenReplica deployments are not extensive and since there exists a fundamental trade-off between optimality and query time which avoids exhaustive search [39], our greedy approach performs an assignment within 3s for a data center with 80,000 hosts, and we later show that the greedy approach achieves fault-tolerance that exceeds that of random placement. of hosts.

3.5 DNS Integration

In an environment where the set of nodes implementing a fault-tolerant object can change at any time, locating the replica set can be a challenge. To help direct clients to the most up-to-date set of replicas, OpenReplica implements special nodes called *name server nodes*. Name server nodes are involved in the underlying Paxos protocol just like replicas, but they maintain no live object and perform no object operations. They solely track meta commands to update the set of live nodes and receive and handle DNS queries.

To support integration with DNS, coordination instances can be assigned a DNS domain, such as `bank.openr.org`, at initialization. On boot, the name server nodes register their IP address and assigned domain with the DNS name servers for their parent domain. Thereafter, the parent domain designates them as authoritative name servers for their subdomain and directs queries accordingly.

Name server nodes also support integration with Amazon Route 53 [3] to enable users to run stand-alone coordination instances without requiring the assistance of a parent domain. To run OpenReplica integrated with Amazon Route 53, the users set up a Route 53 account

that is ready to receive requests and supply the related credentials to OpenReplica. From this point on, the name server nodes track meta commands that affect the view of the system and update the Route 53 account automatically.

DNS integration enables client to initialize their connection to an RSM through a DNS lookup. After the connection is initialized, following method invocations are submitted using the same connection as long as it does not fail. When the connection fails, the client proxy performs a new DNS lookup and initializes a new connection transparently. This way the view changes that might require new connections to be established are masked by the client proxy. Short timeouts on DNS responses ensure that clients do not cache stale DNS results.

3.6 Proxy Generation

OpenReplica clients interact with a coordination object through a client proxy or the web interface provided. Both of these methods use the client proxy, which is automatically generated by OpenReplica through Python reflection and binary rewriting. OpenReplica parses the coordination object, creates the corresponding abstract syntax tree and, following the original structure of the tree, generates a specialized proxy that performs appropriate argument marshalling and unmarshalling as well as execution blocking and unblocking, where needed. OpenReplica also attaches a security token to every proxy to disable unauthorized method invocations on the replicated object, which is generated with the same token.

Clients can use a client proxy with very little modification compared to the invocation of a local object. Due to the replicated nature of the coordination object, the client proxy might throw additional OpenReplica exceptions. The client has to surround such method invocations with an exception handler to catch and address these exceptions, which relate to network errors such as partitioned network.

3.7 Non-deterministic Operations and Side-Effects

During remote method invocations, non-deterministic operations might result in different states on each replica. OpenReplica deals with non-deterministic operations by performing a Paxos agreement on function calls that might result in different states on different replicas. To enable this kind of behavior, the operations with non-deterministic behaviors are detected with a blacklist and, if one of these operations is performed during a method invocation, a new meta command, including the resulting state after the non-deterministic operation, is started by the replica. When this meta command is executed, the instructions following the non-deterministic operation are executed using the state retrieved from the meta command. This ensures that all replicas observe the same non-deterministic choices.

In our current prototype, we identified method invocations in time, random and socket modules to result in non-deterministic results along with dictionary and set operations. In the Python runtime, dictionaries are implemented as hash tables and sets are implemented as open-addressing hash tables, consequently, inserting and removing items can change their order. OpenReplica determines method invocations that make use of these components and simply sorts them to establish a canonical order. Applications wishing to avoid the sort overhead can use their own deterministic data structures.

3.8 Inconsistent Invocations

By default, every method invocation in OpenReplica provides strong consistency. Its slot location in the execution history is the result of an agreement protocol, and its execution is determined by the globally-agreed slot assignment. Because no replica executes a command unless it has seen the entire prefix of commands, the results are guaranteed to be consistent.

But there are certain application-specific instances where this level of consistency is not necessary. When an application needs high performance and can handle inconsistent results, it is possible to provide a best-effort response with drastically lower overhead. For example, a bank account normally requires fully consistent updates, but a user profiler that wants to determine the user's approximate net worth need not go through the full expense of an RSM transaction. To support these cases, OpenReplica provides a low-overhead call for inconsistent method invocation. Such inconsistent calls are performed on any one of the replicas, and it is up to the application programmer to ensure that they execute with no side-effects, as they may be executed at different times on different replicas. OpenReplica does not invoke agreement for such calls, does not record them in object history, and load-balances them uniformly across the set of replicas for performance. As with the consistency relaxation in ZooKeeper, inconsistent invocations in OpenReplica have the potential to provide a significant boost in performance.

3.9 Garbage Collection

Any long-running system based on agreement on a shared history will need to occasionally prune its history in order to avoid running out of memory. In particular, acceptor nodes in OpenReplica keep a record of completely- and partially-decided commands that needs to be compacted periodically. The key to this compaction is the observation that a prefix of history that has been seen by all acceptors and executed by all replicas can be elided safely and replaced with a snapshot of the object. OpenReplica accomplishes this in two main steps. First, a replica takes a snapshot of the coordination ob-

Object	Total Size	OpenReplica Code
Reliable Data Structures		
Queue	12	0
Stack	11	0
Linked List	36	0
Red Black Tree	149	0
Binary Search Tree	61	0
Reliable Synchronization Primitives		
Lock	33	4
Recursive Lock	38	4
Semaphore	29	4
Bounded Semaphore	31	4
Condition Variable	40	2
Barrier	22	2
Reliable Coordination Objects		
Job Manager	50	4
Membership Service	111	4
Shared Log	14	0
Data Storage on Disk	27	0

Table 1: Coordination Object Sizes in LoC. The implementations for reliable versions of data structures, synchronization primitives and custom coordination objects follow exactly from their local, centralized versions, except for the special Blocking and Unblocking Returns required for synchronization primitives.

ject every τ commands, and issues a meta command to garbage collect the state up to this snapshot. This consensus round on a meta command serves three purposes; namely, the garbage collection command is stored in the acceptor nodes; the acceptors detect the meta command and acquiesce only if they themselves have all the ballots for all preceding slot numbers; and finally, the meta command ensures that at the time of execution for the meta command, all the replicas will have the same state. Later, when the meta command is executed, a garbage collection command is sent to acceptor nodes along with the snapshot of the object at that point in time. Upon receiving this message, the acceptors can safely replace a slot with the snapshot of the object and delete old ballot information. This way, during a failover, new leader will be able to simply resurrect the object state after $n\tau$ operations, instead of having to apply as many state transitions.

4 Evaluation

We have performed a detailed evaluation of OpenReplica's performance and compared it to widely used and state-of-the-art coordination service ZooKeeper. In this section we present the size of coordination objects used to implement reliable data structures, synchronization primitives and specialized constructs, results of several microbenchmarks which examine the latency, the recovery time from failures, the throughput and the scalability of OpenReplica, and the performance of the greedy rack-aware placement algorithm used by OpenReplica.

Our experiments reflect end-to-end measurements from clients and include the full overhead of going over the network. As a result, the latency numbers we present are not comparable to numbers presented in most past work, which has tended to report performance metrics collected on the same host. The inputs to clients are generated beforehand and same inputs are used for OpenReplica and ZooKeeper tests.

Our evaluation is performed on a cluster of eleven servers. Each server has two Intel Xeon E5420 processors with 4 cores and a clock speed 2.5 GHz and 16 GB RAM and a 500 GB SATA 3.0 $\mathrm{Gbit/s}$ hard disk operating at 7200 RPM. All servers are running 64-bit Fedora 10 with the Linux 2.6.27 kernel. We spread clients, replicas and acceptors on these 11 servers.

4.1 Implementation Size

The OpenReplica approach results in a great simplification in the implementation of reliable and fault-tolerant coordination objects, including reliable data structures and synchronization primitives. To illustrate, Table 1 presents the sizes of some reliable data structures, synchronization primitives and generalized coordination objects implemented to work with OpenReplica. While implementing these coordination objects, no OpenReplica-specific code has been used except for the Blocking and Unblocking Return exceptions required to implement the multi-return mechanism of synchronization primitives. Consequently, implementing a reliable and fault-tolerant data structure, synchronization primitive or coordination object suitable to be used in a distributed system is reduced to implementing a centralized, local version of it.

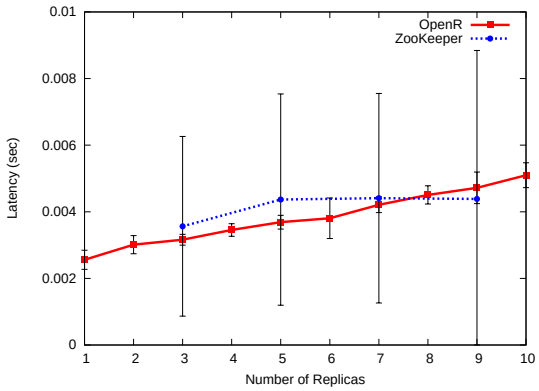


Figure 5: Latency as a function of the number of replicas and acceptors.

4.2 Latency

Next, we examine the latency of consistent requests in OpenReplica and in ZooKeeper. For this experiment, we used a synchronous client that invokes methods from the Account object of Section 2, and collected end-to-end latency measurements from the clients. To be able to examine the latency related to the underlying protocol,

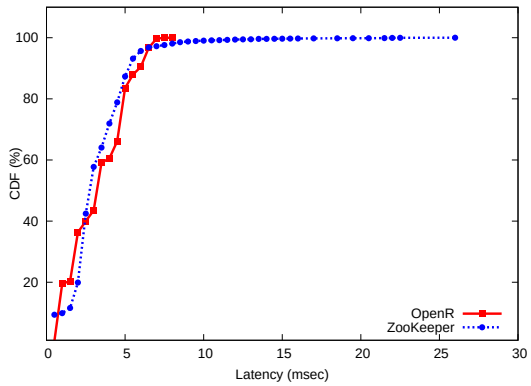


Figure 6: Latency as a function of the number of replicas and acceptors.

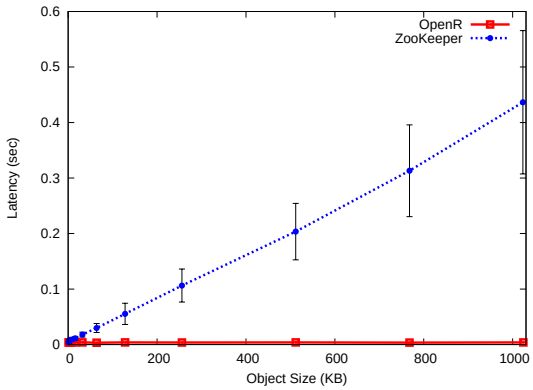


Figure 7: Latency as a function of the size of the replicated state.

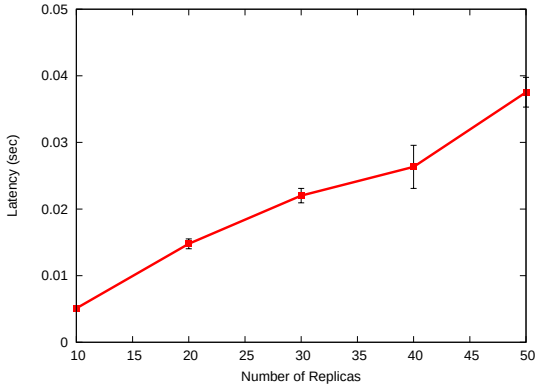


Figure 8: Latency as a function of number of replicas and acceptors.

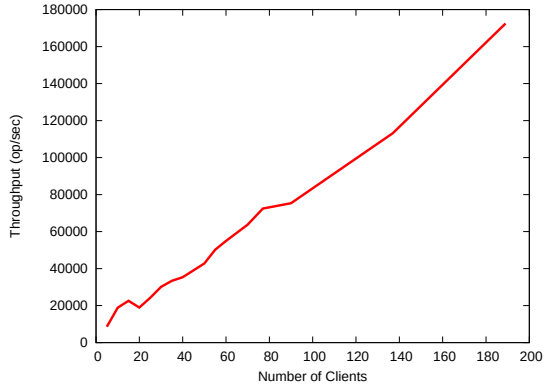


Figure 9: The throughput performance of OpenReplica with 5 replicas and 5 acceptors, for inconsistent reads, against the number of clients.

we used only 128 bytes of replicated state, keeping the serialization and deserialization cost to a minimum for ZooKeeper.

Figure 5 plots the latency of requests against the number of replicas and acceptors in OpenReplica and number of replicas in ZooKeeper. OpenReplica and ZooKeeper present comparable latency results. OpenReplica shows lower latency for lower number of replicas and differs from ZooKeeper performance by 0.5ms on average for larger number of replicas. Another behavior we examine in this graph is the high standard deviation present in ZooKeeper measurements. While OpenReplica requests are handled with the same average latency, there is a big variance in latency measurements for ZooKeeper.

The CDF for OpenReplica and ZooKeeper latency shows this variance in more detail (Figure 6). For clarity, the plots are cutoff at the maximum latency value they present. The CDF of OpenReplica shows that there is an even distribution of latency values shown by OpenReplica varying from values less than 1 ms to 8 ms. ZooKeeper, on the other hand, has a number of requests whose latency exceeds 10 ms, even though these measurements were done when the services were in a stable state with no failures.

4.3 Scalability

We examine the scalability of OpenReplica in relation to size of the replicated state and the number of replicas and acceptors. Figure 7 shows how ZooKeeper and OpenReplica scale as the size of the replicated state grows. As ZooKeeper has to serialize and deserialize data and perform read and write operations with an update on the replicated size, the overhead of these operations increases as the replicated state grows larger. OpenReplica on the other hand, does not require serialization, deserialization or reinstantiation as every object is kept as a live instance. Consequently, the size of the replicated state does not effect the latency experienced by the clients in OpenReplica, providing the same performance for any replicated state size.

A critical parameter in any fault-tolerant system is the amount of fault-tolerance the system offers. Figure 8 shows how OpenReplica scales as the number of replicas and acceptors increase, when the fault-tolerance of the system is improved considerably. The graph shows that OpenReplica performance scales well, even with very large numbers of replicas and acceptors.

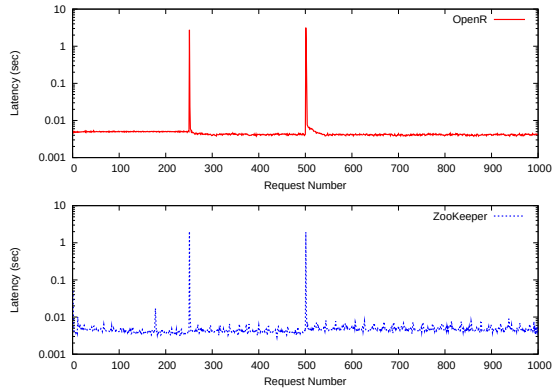


Figure 10: Latency in the presence of leader failures at operation 250 and 500.

4.4 Throughput

OpenReplica achieves a sustained throughput of 327 ops/s with 5 replicas and 5 acceptors; this measurement includes all network overhead. For comparison, ZooKeeper achieves 1872 ops/s in the same setting. The difference is due to an unoptimized OpenReplica implementation in Python, and a highly optimized ZooKeeper implementation that employs batching to improve throughput. This is consistent with the latency measurements for the two systems, where OpenReplica outperforms ZooKeeper because the batching optimizations are not effective for the latency experiment.

OpenReplica implements a fast-read operation that provides high throughput, but inconsistent, method invocations. Fast-read operations do not require agreement, are handled by any replica in the system and are not saved in the command history. ZooKeeper provides a similar read relaxation primitive. Figure 9 examines the throughput of OpenReplica with inconsistent method invocations with 5 replicas. The experiment shows that the throughput scales with increasing numbers of clients. This is not surprising, as inconsistent reads enable OpenReplica to avoid agreement overhead entirely and distribute the request stream among all the replicas.

4.5 Fault Tolerance

Another important performance measure for a fault-tolerant system is how fast the system can recover from the failure of a server, specifically from the failure of the leader. Figure 10 shows how OpenReplica and ZooKeeper handle failures of leaders. In this benchmark, two leaders fail at the 250th and 500th requests, respectively. OpenReplica takes on average 2.75 seconds to recover whereas ZooKeeper takes on average 2 seconds. The recovery performance of OpenReplica depends heavily on the state to be transferred from the acceptors, as a new leader needs to collect all past state from acceptors, constituting the dominant cost of a failover. This overhead, in turn, is determined by the frequency of garbage collection performed in the system; it does not increase with longer amounts of time the system is kept alive.

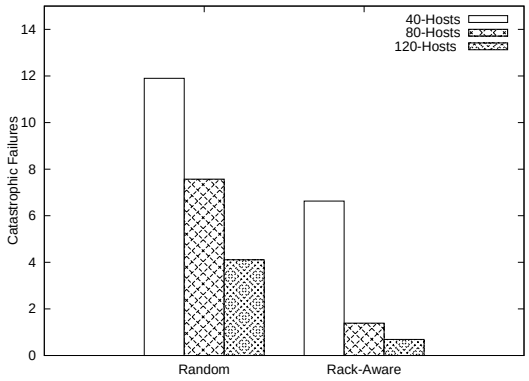


Figure 11: Rack-aware replica placement reduces number of catastrophic failures significantly.

4.6 Rack-Aware Placement

OpenReplica can recover from replica failures if more than f replicas, out of $2f+1$, do not fail concurrently. To prevent these type of *catastrophic failures*, OpenReplica supports placing replica, acceptor and name server nodes in a rack-aware manner. Figure 11 compares the performance of OpenReplica's greedy replica placement strategy to that of random placement. It plots the number of catastrophic failures expected within a year for a system with 5 replicas instatiated on groups of 20-hosts suballocated within a data center with the failure groups shown in Figure 4. This suballocation strategy captures a realistic scenario that a developer at a large company might face when reserving dedicated nodes within a data center, where groups of nodes within a rack are allocated to a project from the larger data center. The probabilities for component failures were extracted from empirical studies [13, 14]. The figure shows that in this setting, greedy placement achieves significant advantages compared to random placement.

5 Related Work

OpenReplica is implemented to provide infrastructure services for distributed systems using the replicated state machine approach [26]. Originally described for fault-free environments, this approach was extended to handle fail-stop failures [42], a class of failures between fail-stop and Byzantine [25], and full Byzantine failures [27]. The seminal tutorial on the state machine approach outlined various implementation strategies for achieving agreement and order requirements [43].

There has been much work examining strategies for achieving the agreement and order requirements in a replicated state machine. In particular, the Paxos Synod

protocol achieves consensus among replicas in an environment with crash failures [28, 29]. Subsequent work has concentrated on making the basic Paxos algorithm more efficient and dynamic [31, 30], two techniques employed by OpenReplica. Other work has concentrated on the practical aspects of implementing the basic algorithm [11, 7, 23, 2, 33]. There has also been some work on designing high performance protocols derived from Paxos [36].

Paxos replicated state machines have been used previously to provide the underlying infrastructure for systems such as consistent, replicated, migratable file systems. SMART [32] achieves high performance through parallelization, supports dynamic membership changes and migration without a window of vulnerability. SMARTER [10] constructs a reliable storage system using Paxos RSMs that are strongly crafted to mask the latencies related to the RSM infrastructure and allows restarts of the system by logging requests. OpenReplica is built to provide a service that enables users to implement such systems easily. While, many systems use Paxos in a monolithic fashion to support a fixed API, OpenReplica is the first system to provide an open, general-purpose object replication service.

Another approach to achieving consistency in a distributed system relies on an atomic broadcast primitive [12]. ZooKeeper follows this approach and implements universal wait-free synchronization primitives [16], which can be used for leader-based atomic broadcast [37, 22]. There have also been similar work on protocols [47, 40] presenting optimistic and collision-fast atomic broadcast protocols, respectively.

Coordination of distributed applications is a long-standing problem and there has been a lot of work focusing on how to provide coordination services for distributed systems and data center environments. Early work examined how to use locks as the basis of coordination among distributed components [24, 18, 34]. More recently, Boxwood [34], designed especially for storage applications, provides reliable data structure abstractions supported directly by the storage infrastructure. Although Boxwood offers a rich set of data structures, its API is not extensible, making it a closed system.

Automatic data center management services have recently emerged to ease the task of managing large-scale distributed systems [15, 20, 1]. Autopilot [20] is a Paxos RSM that handles tasks, such as provisioning, deployment, monitoring and repair, automatically without operator intervention. Similarly, Centrifuge [1] is a lease manager, built on top of a Paxos RSM, that can be used to configure and partition requests among servers. Much like these infrastructure services, OpenReplica is designed to offer a manageable coordination infrastructure that allows programmers to offload complicated configuration and coordination services to an automatically maintained, fault-tolerant and available service.

Past work on toolkits for replication services examined how to build infrastructure services. PRACTI [5] approach offers partial replication of state on different nodes, arbitrary consistency guarantees on the replicated data and arbitrary messaging between replicas. Ursa [6] offers safety and liveness policies that provide different consistency levels for a replicated system, and provides mechanisms that define abstractions for storage, communication, and consistency. This enables Ursa to be used as infrastructure for higher-level replication systems. In contrast to such low-level services for the construction of replication systems, OpenReplica provides a higher abstraction that directly replicates user objects.

Past work has examined how to employ an object-oriented distributed programming (OODP) paradigm. Common Object Request Broker Architecture (CORBA) [44] provides an open standard for OODP, providing a mechanism to normalize method invocations among different platforms. There has been much work on building mechanisms for distributed systems using CORBA [19, 41] and extending CORBA to provide additional guarantees such as fault-tolerance [35]. A similar approach was used to build distributed objects that remain available and offer guarantees on the completion of operations in the presence of up to k failures [9]. This work was later used to provide a platform independent framework for fault-tolerance [46]. While OpenReplica shares the same object-oriented spirit as these early efforts, it differs fundamentally in every aspect of its implementation.

6 Conclusions

This paper presented OpenReplica, an object-oriented coordination service for large-scale distributed systems. OpenReplica proposes a novel approach to providing replication and synchronization in large-scale distributed systems. This approach is based around the abstraction of a coordination object; namely, an object that defines a replicated state machine that can block and resume the execution of its clients. Coordination objects not only support ordinary replication, but also enable complex distributed synchronization constructs and reliable data structures. Critically, OpenReplica renders the specification of such constructs straightforward and similar to their non-distributed counterparts.

In contrast with the file-base APIs of extant coordination services, OpenReplica object-based API represents a novel approach to replica management. Whereas previous systems provide low-level mechanisms that could be used in a large number of ways to implement replicated state machines, OpenReplica provides a high-level approach. These state machines are specified using regular

Python objects. OpenReplica maintains a live instance of these coordination objects on every replica node and uses Paxos to guarantee strong consistency in the presence of crash failures. Moreover, OpenReplica implements additional mechanisms to guarantee the soundness of the replicated state in the presence of non-deterministic invocations and side effects. Evaluations show that OpenReplica provides performance, in terms of latency, scalability and failover, that is comparable to ZooKeeper, while providing additional features as well as a higher level of abstraction.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, Berkeley, CA, USA, 2010.

[2] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I Do Declare: Consensus in a Logic Language. *SIGOPS Operating Systems Review*, 43:25–30, ACM, New York, NY, January 2010.

[3] Amazon Web Services LLC. Amazon Route 53. `http://aws.amazon.com/route53/`, Accessed 2012 2, May.

[4] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350, USENIX Association, Berkeley, CA, 2006.

[5] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI Replication. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[6] N. Belaramani, M. Dahlin, A. Nayate, and J. Zheng. Making Replication Simple with Ursa. University of Texas at Austin Department of Computer Sciences, Technical Report TR-07-57, Oct 2007.

[7] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34:47–67, ACM, New York, NY, March 2003.

[8] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2:39–59, ACM, New York, NY, February 1984.

[9] K. P. Birman, a. T. A. Joseph, T. Raeuchle, and A. E. Abbadi. Implementing Fault-Tolerant Distributed Objects. *IEEE Transactions on Software Engineering*, 11:502–508, IEEE Press, Piscataway, NJ, USA, June 1985.

[10] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, Berkeley, CA, USA, 2011.

[11] T. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. *In Proceedings of PODC*, pages 398–407, ACM Press, 2007.

[12] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43:225–267, ACM, New York, NY, March 1996.

[13] J. Dean. Software Engineering Advice from Building Large-Scale Distributed Systems. `http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf`, Accessed 2012 2, May.

[14] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, ACM, New York, NY, USA, 2011.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–8, USENIX Association, Berkeley, CA, USA, 2010.

[16] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, ACM, New York, NY, January 1991.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, 2010.

[18] A. B. Hastings. Distributed Lock Management in a Transaction Processing Environment. *IEEE 9th Symposium on Reliable Distributed Systems*, pages 22–31, October 1990.

[19] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Sesign and Performance of a Real-time CORBA Event Service. *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 184–200, ACM, New York, NY, USA, 1997.

[20] M. Isard. Autopilot: Automatic Data Center Management. *SIGOPS Operating Systems Review*, 41(2):60–67, ACM, New York, NY, USA, 2007.

[21] F. P. Junqueira and K. Marzullo. Synchronous Consensus for Dependent Process Failures. *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 274– IEEE Computer Society, Washington, DC, USA, 2003.

[22] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-Performance Broadcast for Primary-Backup Systems. *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 245–256, June 2011.

[23] J. Kirsch and Y. Amir. Paxos for System Builders: An Overview. *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 3:1–3:6, ACM, New York, NY, 2008.

[24] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. VAXcluster: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4:130–146, ACM, New York, NY, May 1986.

[25] L. Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks (1976)*, 2(2):95–114, 1978.

[26] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of ACM*, 21:558–565, ACM, New York, NY, July 1978.

[27] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 6:254–280, 1984.

[28] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[29] L. Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[30] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[31] L. Lamport and M. Massa. Cheap Paxos. *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, IEEE Computer Society, Washington, DC, 2004.

[32] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The SMART Way to Migrate Replicated Stateful Services. *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 103–115, ACM, New York, NY, USA, 2006.

[33] D. Mazières. Paxos Made Practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, Accessed May 2, 2012.

[34] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. *Proceedings of the 6th Conference on Symposium*

on Opearting Systems Design and Implementation - Volume 6, USENIX Association, Berkeley, CA, 2004.

[35] S. Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, USENIX Association, Berkeley, CA, USA, 1995.

[36] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 369–384, USENIX Association, Berkeley, CA, 2008.

[37] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 2:1–2:6, ACM, New York, NY, 2008.

[38] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and its Uses. *SIGCOMM Computer Communication Review*, 35:73–84, ACM, New York, NY, August 2005.

[39] D. Spence, J. Crowcroft, S. Hand, and T. Harris. Location Based Placement of Whole Distributed Systems. *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology*, pages 124–134, ACM, New York, NY, USA, 2005.

[40] R. Schmidt, L. Camargos, and F. Pedone. On Collision-Fast Atomic Broadcast. EPFL, 2007.

[41] D. C. Schmidt, A. S. Gokhale, T. H. Harrison, and G. Parulkar. A High-Performance End System Architecture for Real-Time CORBA. *Communications Magazine, IEEE*, 35(2):72–77, February 1997.

[42] F. B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Language Systems*, 4:125–148, ACM, New York, NY, April 1982.

[43] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22:299–319, ACM, New York, NY, December 1990.

[44] S. Vinoski. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *Communications Magazine, IEEE*, 35(2):46–55, February 1997.

[45] R. van Renesse. Paxos Made Moderately Complex. `http://www.cs.cornell.edu/courses/CS7412/2011sp/paxos.pdf`, Accessed May 2, 2012.

[46] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39:76–83, ACM, New York, NY, USA, April 1996.

[47] P. Zielinski. Low-Latency Atomic Broadcast in the Presence of Contention. *Distributed Computing*, 20(6):435–450, 2008.