In *Proc. USENIX Annual Technical Conference*.
San Diego, CA, June 2009.

Object Storage on CRAQ

High-throughput chain replication for read-mostly workloads

Jeff Terrace and Michael J. Freedman
Princeton University

Abstract

Massive storage systems typically replicate and partition data over many potentially-faulty components to provide both reliability and scalability. Yet many commercially-deployed systems, especially those designed for interactive use by customers, sacrifice stronger consistency properties in the desire for greater availability and higher throughput.

This paper describes the design, implementation, and evaluation of CRAQ, a distributed object-storage system that challenges this inflexible tradeoff. Our basic approach, an improvement on Chain Replication, maintains strong consistency while greatly improving read throughput. By distributing load across all object replicas, CRAQ scales linearly with chain size without increasing consistency coordination. At the same time, it exposes non-committed operations for weaker consistency guarantees when this suffices for some applications, which is especially useful under periods of high system churn. This paper explores additional design and implementation considerations for geo-replicated CRAQ storage across multiple datacenters to provide locality-optimized operations. We also discuss multi-object atomic updates and multicast optimizations for large-object updates.

1 Introduction

Many online services require *object-based* storage, where data is presented to applications as entire units. Object stores support two basic primitives: *read* (or query) operations return the data block stored under an object name, and *write* (or update) operations change the state of a single object. Such object-based storage is supported by key-value databases (*e.g.*, BerkeleyDB [40] or Apache's semi-structured CouchDB [13]) to the massively-scalable systems being deployed in commercial datacenters (*e.g.*, Amazon's Dynamo [15], Facebook's Cassandra [16], and the popular Memcached [18]). To achieve the requisite reliability, load balancing, and scalability in many of these systems, the object namespace is partitioned over many machines and each data object is replicated several times.

Object-based systems are more attractive than their file-system counterparts when applications have certain requirements. Object stores are better suited for flat namespaces, such as in key-value databases, as opposed to hierarchical directory structures. Object stores simplify the process of supporting whole-object modifications. And, they typically only need to reason about the ordering of modifications *to a specific object*, as opposed to the entire storage system; it is significantly cheaper to provide consistency guarantees per object instead of across all operations and/or objects.

When building storage systems that underlie their myriad applications, commercial sites place the need for high performance and availability at the forefront. Data is replicated to withstand the failure of individual nodes or even entire datacenters, whether from planned maintenance or unplanned failure. Indeed, the news media is rife with examples of datacenters going offline, taking down entire websites in the process [26]. This strong focus on availability and performance—especially as such properties are being codified in tight SLA requirements [4, 24]—has caused many commercial systems to sacrifice *strong consistency* semantics due to their perceived costs (as at Google [22], Amazon [15], eBay [46], and Facebook [44], among others).

Recently, van Renesse and Schneider presented a *chain replication* method for object storage [47] over fail-stop servers, designed to provide strong consistency yet improve throughput. The basic approach organizes all nodes storing an object in a chain, where the chain tail handles all read requests, and the chain head handles all write requests. Writes propagate down the chain before the client is acknowledged, thus providing a simple ordering of all object operations—and hence strong consistency—at the tail. The lack of any complex or multi-round protocols yields simplicity, good throughput, and easy recovery.

Unfortunately, the basic chain replication approach has some limitations. All reads for an object must go to the same node, leading to potential hotspots. Multiple chains can be constructed across a cluster of nodes for better load balancing—via consistent hashing [29] or a more centralized directory approach [22]—but these algorithms might still find load imbalances if particular objects are disproportionally popular, a real issue in practice [17]. Perhaps an even more serious issue arises when attempting to build chains across multiple datacenters, as all reads to a chain may then be handled by a potentially-distant node (the chain's tail).

This paper presents the design, implementation, and evaluation of CRAQ (*Chain Replication with Apportioned Queries*), an object storage system that, while maintaining the strong consistency properties of chain replication [47], provides lower latency and higher throughput for read operations by supporting *apportioned queries*: that is, dividing read operations over all nodes in a chain, as opposed to requiring that they all be handled by a single primary node. This paper's main contributions are the following.

1. CRAQ enables any chain node to handle read operations while preserving strong consistency, thus supporting load balancing across all nodes storing an object. Furthermore, when workloads are read mostly—an assumption used in other systems such as the Google File System [22] and Memcached [18]—the performance of CRAQ rivals systems offering only eventual consistency.

2. In addition to strong consistency, CRAQ's design naturally supports eventual-consistency among read operations for lower-latency reads during write contention and degradation to read-only behavior during transient partitions. CRAQ allows applications to specify the maximum staleness acceptable for read operations.

3. Leveraging these load-balancing properties, we describe a wide-area system design for building CRAQ chains across geographically-diverse clusters that preserves strong locality properties. Specifically, reads can be handled either completely by a local cluster, or at worst, require concise metadata information to be transmitted across the wide-area during times of high write contention. We also present our use of ZooKeeper [48], a PAXOS-like group membership system, to manage these deployments.

Finally, we discuss additional extensions to CRAQ, including the integration of mini-transactions for multi-object atomic updates, and the use of multicast to improve write performance for large-object updates. We have not yet finished implementing these optimizations, however.

A preliminary performance evaluation of CRAQ demonstrates its high throughput compared to the basic chain replication approach, scaling linearly with the number of chain nodes for read-mostly workloads: approximately a 200% improvement for three-node chains, and 600% for seven-node chains. During high write contention, CRAQ's read throughput in three-node chains still outperformed chain replication by a factor of two, and read latency remains low. We characterize its performance under varying workloads and under failures. Finally, we evaluate CRAQ's performance for geo-replicated storage, demonstrating significantly lower latency than that achieved by basic chain replication.

The remainder of this paper is organized as follows. Section §2 provides a comparison between the basic chain replication and CRAQ protocols, as well as CRAQ's support for eventual consistency. Section §3 describes scaling out CRAQ to many chains, within and across datacenters, as well as the group membership service that manages chains and nodes. Section §4 touches on extensions such as multi-object updates and leveraging multicast. Section §5 describes our CRAQ implementation, §6 presents our performance evaluation, §7 reviews related work, and §8 concludes.

2 Basic System Model

This section introduces our object-based interface and consistency models, provides a brief overview of the standard Chain Replication model, and then presents strongly-consistent CRAQ and its weaker variants.

2.1 Interface and Consistency Model

An object-based storage system provides two simple primitives for users:

- ***write(objID, V)***: The write (update) operation stores the value V associated with object identifier $objID$.
- ***V ← read(objID)***: The read (query) operation retrieves the value V associated with object id $objID$.

We will be discussing two main types of consistency, taken with respect to individual objects.

- **Strong Consistency** in our system provides the guarantee that all read and write operations to an object are executed in some sequential order, and that a read to an object always sees the latest written value.
- **Eventual Consistency** in our system implies that writes to an object are still applied in a sequential order on all nodes, but eventually-consistent reads to different nodes can return stale data for some period of inconsistency (*i.e.*, before writes are applied on all nodes). Once all replicas receive the write, however, read operations will never return an older version than this latest committed write. In fact, a client will also see monotonic read consistency[1] if it main-

[1] That is, informally, successive reads to an object will return either the same prior value or a more recent one, but never an older value.

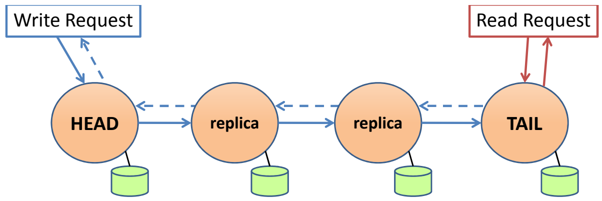

Figure 1: **All reads in Chain Replication must be handled by the tail node, while all writes propagate down the chain from the head.**

tains a session with a particular node (although not across sessions with different nodes).

We next consider how Chain Replication and CRAQ provide their strong consistency guarantees.

2.2 Chain Replication

Chain Replication (CR) is a method for replicating data across multiple nodes that provides a strongly consistent storage interface. Nodes form a *chain* of some defined length C. The *head* of the chain handles all *write* operations from clients. When a write operation is received by a node, it is propagated to the next node in the chain. Once the write reaches the tail node, it has been applied to all replicas in the chain, and it is considered *committed*. The tail node handles all *read* operations, so only values which are committed can be returned by a *read*.

Figure 1 provides an example chain of length four. All read requests arrive and are processed at the tail. Write requests arrive at the head of the chain and propagate their way down to the tail. When the tail commits the write, a reply is sent to the client. The CR paper describes the tail sending a message directly back to the client; because we use TCP, our implementation actually has the head respond after it receives an acknowledgment from the tail, given its pre-existing network connection with the client. This acknowledgment propagation is shown with the dashed line in the figure.

The simple topology of CR makes write operations cheaper than in other protocols offering strong consistency. Multiple concurrent writes can be pipelined down the chain, with transmission costs equally spread over all nodes. The simulation results of previous work [47] showed competitive or superior throughput for CR compared to primary/backup replication, while arguing a principle advantage from quicker and easier recovery.

Chain replication achieves strong consistency: As all reads go to the tail, and all writes are committed only when they reach the tail, the chain tail can trivially apply a total ordering over all operations. This does come at a cost, however, as it reduces read throughput to that of a single node, instead of being able to scale out with chain size. But it is necessary, as querying intermediate nodes could otherwise violate the strong consistency guarantee; specifically, concurrent reads to different nodes could see different writes as they are in the process of propagating down the chain.

While CR focused on providing a storage service, one could also view its query/update protocols as an interface to replicated state machines (albeit ones that affect distinct object). One can view CRAQ in a similar light, although the remainder of this paper considers the problem only from the perspective of a read/write (also referred to as a get/put or query/update) object storage interface.

2.3 Chain Replication with Apportioned Queries

Motivated by the popularity of read-mostly workload environments, CRAQ seeks to increase read throughput by allowing any node in the chain to handle read operations while still providing strong consistency guarantees. The main CRAQ extensions are as follows.

1. A node in CRAQ can store multiple versions of an object, each including a monotonically-increasing version number and an additional attribute whether the version is *clean* or *dirty*. All versions are initially marked as clean.

2. When a node receives a new version of an object (via a write being propagated down the chain), the node appends this latest version to its list for the object.

   - If the node is not the tail, it marks the version as dirty, and propagates the write to its successor.
   - Otherwise, if the node is the tail, it marks the version as clean, at which time we call the object version (write) as *committed*. The tail node can then notify all other nodes of the commit by sending an acknowledgement backwards through the chain.

3. When an *acknowledgment* message for an object version arrives at a node, the node marks the object version as clean. The node can then delete all prior versions of the object.

4. When a node receives a read request for an object:

   - If the latest known version of the requested object is clean, the node returns this value.
   - Otherwise, if the latest version number of the object requested is dirty, the node contacts the tail and asks for the tail's last committed version number (a *version query*). The node then returns that version of the object; by construction, the node is guaranteed to be storing this

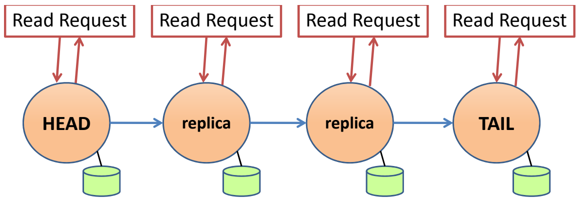

Figure 2: **Reads to clean objects in CRAQ can be completely handled by any node in the system.**

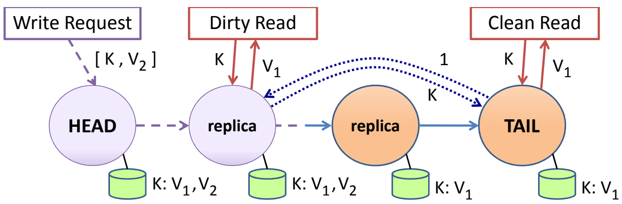

Figure 3: **Reads to dirty objects in CRAQ can be received by any node, but require small version requests (dotted blue line) to the chain tail to properly serialize operations.**

version of the object. We note that although the tail could commit a new version between when it replied to the version request and when the intermediate node sends a reply to the client, this does not violate our definition of strong consistency, as *read operations are serialized with respect to the tail*.

Note that an object's "dirty" or "clean" state at a node can also be determined implicitly, provided a node deletes old versions as soon as it receives a write commitment acknowledgment. Namely, if the node has exactly one version for an object, the object is implicitly in the clean state; otherwise, the object is dirty and the properly-ordered version must be retrieved from the chain tail.

Figure 2 shows a CRAQ chain in the starting clean state. Each node stores an identical copy of an object, so any read request arriving at any node in the chain will return the same value. All nodes remain in the clean state unless a write operation is received.[2]

In Figure 3, we show a write operation in the middle of propagation (shown by the dashed purple line). The head node received the initial message to write a new version (V_2) of the object, so the head's object is dirty. It then propagated the write message down the chain to the second node, which also marked itself as dirty for that object (having multiple versions $[V_1, V_2]$ for a single object ID K). If a read request is received by one of the clean nodes, they immediately return the old version of the object: This is correct, as the new version has yet to be committed at the tail. If a read request is received by either of the dirty nodes, however, they send a version query to the tail—shown in the figure by the dotted blue arrow—which returns its known version number for the requested object (1). The dirty node then returns the old object value (V_1) associated with this specified version number. Therefore, all nodes in the chain will still return the same version of an object, even in the face of multiple outstanding writes being propagated down the chain.

When the tail receives and accepts the write request, it sends an acknowledgment message containing this write's version number back up the chain. As each predecessor receives the acknowledgment, it marks the specified version as clean (possibly deleting all older versions). When its latest-known version becomes clean, it can subsequently handle reads locally. This method leverages the fact that writes are all propagated serially, so the tail is always the last chain node to receive a write.

CRAQ's throughput improvements over CR arise in two different scenarios:

- **Read-Mostly Workloads** have most of the read requests handled solely by the $C-1$ non-tail nodes (as clean reads), and thus throughput in these scenarios scales linearly with chain size C.

- **Write-Heavy Workloads** have most read requests to non-tail nodes as dirty, thus require version queries to the tail. We suggest, however, that these version queries are lighter-weight than full reads, allowing the tail to process them at a much higher rate before it becomes saturated. This leads to a total read throughput that is still higher than CR.

Performance results in §6 support both of these claims, even for small objects. For longer chains that are persistently write-heavy, one could imagine optimizing read throughput by having the tail node *only* handle version queries, not full read requests, although we do not evaluate this optimization.

2.4 Consistency Models on CRAQ

Some applications may be able to function with weaker consistency guarantees, and they may seek to avoid the performance overhead of version queries (which can be significant in wide-area deployments, per §3.3), or they may wish to continue to function at times when the system cannot offer strong consistency (*e.g.*, during partitions). To support such variability in requirements, CRAQ simultaneously supports three different consistency models for

[2] There's a small caveat about the system ordering properties for clean reads. In traditional Chain Replication, all operations are handled by the tail, so it explicitly defines a total ordering over all operations affecting an object. In CRAQ, clean read operations to different nodes are executed locally; thus, while one could define an (arbitrary) total ordering over these "concurrent" reads, the system does not do such explicitly. Of course, both systems explicitly maintain (at the tail) a total ordering with respect to all read/write, write/read, and write/write relationships.

reads. A read operation is annotated with which type of consistency is permissive.

- **Strong Consistency** (the default) is described in the model above (§2.1). All object reads are guaranteed to be consistent with the last committed write.
- **Eventual Consistency** allows read operations to a chain node to return the newest object version known to it. Thus, a subsequent read operation to a different node may return an object version older than the one previously returned. This does not, therefore, satisfy monotonic read consistency, although reads to a single chain node do maintain this property locally (*i.e.*, as part of a *session*).
- **Eventual Consistency with Maximum-Bounded Inconsistency** allows read operations to return newly written objects before they commit, but only to a certain point. The limit imposed can be based on time (relative to a node's local clock) or on absolute version numbers. In this model, a value returned from a read operation is guaranteed to have a maximum inconsistency period (defined over time or versioning). If the chain is still available, this inconsistency is actually in terms of the returned version being *newer* than the last committed one. If the system is partitioned and the node cannot participate in writes, the version may be *older* than the current committed one.

2.5 Failure Recovery in CRAQ

As the basic structure of CRAQ is similar to CR, CRAQ uses the same techniques to recover from failure. Informally, each chain node needs to know its predecessor and successor, as well as the chain head and tail. When a head fails, its immediate successor takes over as the new chain head; likewise, the tail's predecessor takes over when the tail fails. Nodes joining or failing from within the middle of the chain must insert themselves between two nodes, much like a doubly-linked list. The proofs of correctness for dealing with system failures are similar to CR; we avoid them here due to space limitations. Section §5 describes the details of failure recovery in CRAQ, as well as the integration of our coordination service. In particular, CRAQ's choice of allowing a node to join anywhere in a chain (as opposed only to at its tail [47]), as well as properly handling failures during recovery, requires some careful consideration.

3 Scaling CRAQ

In this section, we discuss how applications can specify various chain layout schemes in CRAQ, both within a single datacenter and across multiple datacenters. We then describe how to use a coordination service to store the chain metadata and group membership information.

3.1 Chain Placement Strategies

Applications that use distributed storage services can be diverse in their requirements. Some common situations that occur may include:

- Most or all writes to an object might originate in a single datacenter.
- Some objects may be only relevant to a subset of datacenters.
- Popular objects might need to be heavily replicated while unpopular ones can be scarce.

CRAQ provides flexible chain configuration strategies that satisfy these varying requirements through the use of a two-level naming hierarchy for objects. An object's identifier consists of both a *chain identifier* and a *key identifier*. The chain identifier determines which nodes in CRAQ will store all keys within that chain, while the key identifier provides unique naming per chain. We describe multiple ways of specifying application requirements:

1. **Implicit Datacenters & Global Chain Size:**

   $$\{num\_datacenters,\ chain\_size\}$$

   In this method, the number of datacenters that will store the chain is defined, but not explicitly which datacenters. To determine exactly which datacenters store the chain, consistent hashing is used with unique datacenter identifiers.

2. **Explicit Datacenters & Global Chain Size:**

   $$\{chain\_size,\ dc_1,\ dc_2,\ \ldots,\ dc_N\}$$

   Using this method, every datacenter uses the same chain size to store replicas within the datacenter. The head of the chain is located within datacenter dc_1, the tail of the chain is located within datacenter dc_N, and the chain is ordered based on the provided list of datacenters. To determine which nodes within a datacenter store objects assigned to the chain, consistent hashing is used on the chain identifier. Each datacenter dc_i has a node which connects to the tail of datacenter dc_{i-1} and a node which connects to the head of datacenter dc_{i+1}, respectively. An additional enhancement is to allow *chain_size* to be 0 which indicates that the chain should use *all* nodes within each datacenter.

3. **Explicit Datacenter Chain Sizes:**

   $$\{dc_1,\ chain\_size_1,\ \ldots,\ dc_N,\ chain\_size_N\}$$

Here the chain size within each datacenter is specified separately. This allows for non-uniformity in chain load balancing. The chain nodes within each datacenter are chosen in the same manner as the previous method, and $chain\_size_i$ can also be set to 0.

In methods 2 and 3 above, dc_1 can be set as a **master datacenter**. If a datacenter is the master for a chain, this means that writes to the chain will only be accepted by that datacenter during transient failures. Otherwise, if dc_1 is disconnected from the rest of the chain, dc_2 could become the new head and take over write operations until dc_1 comes back online. When a master is not defined, writes will only continue in a partition if the partition contains a majority of the nodes in the global chain. Otherwise, the partition will become read-only for maximum-bounded inconsistent read operations, as defined in Section 2.4.

CRAQ could easily support other more complicated methods of chain configuration. For example, it might be desirable to specify an explicit backup datacenter which only participates in the chain if another datacenter fails. One could also define a set of datacenters (*e.g.*, "East coast"), any one of which could fill a single slot in the ordered list of datacenters of method 2. For brevity, we do not detail more complicated methods.

There is no limit on the number of key identifiers that can be written to a single chain. This allows for highly flexible configuration of chains based on application needs.

3.2 CRAQ within a Datacenter

The choice of how to distribute multiple chains across a datacenter was investigated in the original Chain Replication work. In CRAQ's current implementation, we place chains within a datacenter using consistent hashing [29, 45], mapping potentially many chain identifiers to a single head node. This is similar to a growing number of datacenter-based object stores [15, 16]. An alternative approach, taken by GFS [22] and promoted in CR [47], is to use the membership management service as a directory service in assigning and storing randomized chain membership, *i.e.*, each chain can include some random set of server nodes. This approach improves the potential for parallel system recovery. It comes at the cost, however, of increased centralization and state. CRAQ could easily use this alternative organizational design as well, but it would require storing more metadata information in the coordination service.

3.3 CRAQ Across Multiple Datacenters

CRAQ's ability to read from any node improves its latency when chains stretch across the wide-area: When clients have flexibility in their choice of node, they can choose one that is nearby (or even lightly loaded). As long as the chain is clean, the node can return its local replica of an object without having to send any wide-area requests. With traditional CR, on the other hand, all reads would need to be handled by the potentially-distant tail node. In fact, various designs may choose head and/or tail nodes in a chain based on their datacenter, as objects may experience significant reference locality. Indeed, the design of PNUTS [12], Yahoo!'s new distributed database, is motivated by the high write locality observed in their datacenters.

That said, applications might further optimize the selection of wide-area chains to minimize write latency and reduce network costs. Certainly the naive approach of building chains using consistent hashing across the entire global set of nodes leads to randomized chain successors and predecessors, potentially quite distant. Furthermore, an individual chain may cross in and out of a datacenter (or particular cluster within a datacenter) several times. With our chain optimizations, on the other hand, applications can minimize write latency by carefully selecting the order of datacenters that comprise a chain, and we can ensure that a single chain crosses the network boundary of a datacenter only once in each direction.

Even with an optimized chain, the latency of write operations over wide-area links will increase as more datacenters are added to the chain. Although this increased latency could be significant in comparison to a primary/backup approach which disseminates writes in parallel, it allows writes to be pipelined down the chain. This vastly improves write throughput over the primary/backup approach.

3.4 ZooKeeper Coordination Service

Building a fault-tolerant coordination service for distributed applications is notoriously error prone. An earlier version of CRAQ contained a very simple, centrally-controlled coordination service that maintained membership management. We subsequently opted to leverage ZooKeeper [48], however, to provide CRAQ with a robust, distributed, high-performance method for tracking group membership and an easy way to store chain metadata. Through the use of Zookeper, CRAQ nodes are guaranteed to receive a notification when nodes are added to or removed from a group. Similarly, a node can be notified when metadata in which it has expressed interest changes.

ZooKeeper provides clients with a hierarchical namespace similar to a filesystem. The filesystem is stored in memory and backed up to a log at each ZooKeeper instance, and the filesystem state is replicated across multiple ZooKeeper nodes for reliability and scalability. To reach agreement, ZooKeeper nodes use an atomic broadcast protocol similar to two-phase-commit. Optimized for read-mostly, small-sized workloads, ZooKeeper provides good performance in the face of many readers since it can serve the majority of requests from memory.

Similar to traditional filesystem namespaces, ZooKeeper clients can list the contents of a directory, read the value associated with a file, write a value to a file, and receive a notification when a file or directory is modified or deleted. ZooKeeper's primitive operations allow clients to implement many higher-level semantics such as group membership, leader election, event notification, locking, and queuing.

Membership management and chain metadata across multiple datacenters does introduce some challenges. In fact, ZooKeeper is not optimized for running in a multi-datacenter environment: Placing multiple ZooKeeper nodes within a single datacenter improves Zookeeper read scalability within that datacenter, but at the cost of wide-area performance. Since the vanilla implementation has no knowledge of datacenter topology or notion of hierarchy, coordination messages between Zookeeper nodes are transmitted over the wide-area network multiple times. Still, our current implementation ensures that CRAQ nodes always receive notifications from *local* Zookeeper nodes, and they are further notified only about chains and node lists that are relevant to them. We expand on our coordination through Zookeper in §5.1.

To remove the redundancy of cross-datacenter ZooKeeper traffic, one could build a hierarchy of Zookeeper instances: Each datacenter could contain its own local ZooKeeper instance (of multiple nodes), as well as having a representative that participates in the global ZooKeeper instance (perhaps selected through leader election among the local instance). Separate functionality could then coordinate the sharing of data between the two. An alternative design would be to modify ZooKeeper itself to make nodes aware of network topology, as CRAQ currently is. We have yet to fully investigate either approach and leave this to future work.

4 Extensions

This section discusses some additional extensions to CRAQ, including its facility with mini-transactions and the use of multicast to optimize writes. We are currently in the process of implementing these extensions.

4.1 Mini-Transactions on CRAQ

The whole-object read/write interface of an object store may be limiting for some applications. For example, a BitTorrent tracker or other directory service would want to support list addition or deletion. An analytics service may wish to store counters. Or applications may wish to provide conditional access to certain objects. None of these are easy to provide only armed with a pure object-store interface as described so far, but CRAQ provides key extensions that support transactional operations.

4.1.1 Single-Key Operations

Several single-key operations are trivial to implement, which CRAQ already supports:

- **Prepend/Append:** Adds data to the beginning or end of an object's current value.
- **Increment/Decrement:** Adds or subtracts to a key's object, interpreted as an integer value.
- **Test-and-Set:** Only update a key's object if its current version number equals the version number specified in the operation.

For Prepend/Append and Increment/Decrement operations, the head of the chain storing the key's object can simply apply the operation to the latest version of the object, even if the latest version is dirty, and then propagate a full replacement write down the chain. Furthermore, if these operations are frequent, the head can buffer the requests and batch the updates. These enhancements would be much more expensive using a traditional two-phase-commit protocol.

For the test-and-set operation, the head of the chain checks if its most recent committed version number equals the version number specified in the operation. If there are no outstanding uncommitted versions of the object, the head accepts the operation and propagates an update down the chain. If there are outstanding writes, we simply reject the test-and-set operation, and clients are careful to back off their request rate if continuously rejected. Alternatively, the head could "lock" the object by disallowing writes until the object is clean and recheck the latest committed version number, but since it is very rare that an uncommitted write is aborted and because locking the object would significantly impact performance, we chose not to implement this alternative.

The test-and-set operation could also be designed to accept a value rather than a version number, but this introduces additional complexity when there are outstanding uncommitted versions. If the head compares against the most recent committed version of the object (by contacting the tail), any writes that are currently in progress would not be accounted for. If instead the head compares against the most recent uncommitted version, this violates consistency guarantees. To achieve consistency, the head would need to temporarily lock the object by disallowing (or temporarily delaying) writes until the object is clean. This does not violate consistency guarantees and ensures that no updates are lost, but could significantly impact write performance.

4.1.2 Single-Chain Operations

Sinfonia's recently proposed "mini-transactions" provide an attractive lightweight method [2] of performing transactions on multiple keys within a single chain. A mini-transaction is defined by a compare, read, and write set; Sinfonia exposes a linear address space across many memory nodes. A compare set tests the values of the specified address location and, if they match the provided values, executes the read and write operations. Typically designed for settings with low write contention, Sinfonia's mini-transactions use an optimistic two-phase commit protocol. The prepare message attempts to grab a lock on each specified memory address (either because different addresses were specified, or the same address space is being implemented on multiple nodes for fault tolerance). If all addresses can be locked, the protocol commits; otherwise, the participant releases all locks and retries later.

CRAQ's chain topology has some special benefits for supporting similar mini-transactions, as applications can designate multiple objects be stored on the same chain—*i.e.*, those that appear regularly together in multi-object mini-transactions—in such a way that preserves locality. Objects sharing the same *chainid* will be assigned the same node as their chain head, reducing the two-phase commit to a single interaction because only one head node is involved. CRAQ is unique in that mini-transactions that only involve a single chain can be accepted using only the single head to mediate access, as it controls write access to all of a chain's keys, as opposed to all chain nodes. The only trade-off is that write throughput may be affected if the head needs to wait for keys in the transaction to become clean (as described in §4.1.1). That said, this problem is only worse in Sinfonia as it needs to wait (by exponentially backing off the mini-transaction request) for unlocked keys across multiple nodes. Recovery from failure is similarly easier in CRAQ as well.

4.1.3 Multi-Chain Operations

Even when multiple chains are involved in multi-object updates, the optimistic two-phase protocol need only be implemented with the chain heads, not all involved nodes. The chain heads can lock any keys involved in the mini-transaction until it is fully committed.

Of course, application writers should be careful with the use of extensive locking and mini-transactions: They reduce the write throughput of CRAQ as writes to the same object can no longer be pipelined, one of the very benefits of chain replication.

4.2 Lowering Write Latency with Multicast

CRAQ can take advantage of multicast protocols [41] to improve write performance, especially for large updates or long chains. Since chain membership is stable between node membership changes, a multicast group can be created for each chain. Within a datacenter, this would probably take the form of a network-layer multicast protocol, while application-layer multicast protocols may be better-suited for wide-area chains. No ordering or reliability guarantees are required from these multicast protocols.

Then, instead of propagating a full write serially down a chain, which adds latency proportional to the chain length, the actual value can be multicast to the entire chain. Then, only a small metadata message needs to be propagated down the chain to ensure that all replicas have received a write before the tail. If a node does not receive the multicast for any reason, the node can fetch the object from its predecessor after receiving the write commit message and before further propagating the commit message.

Additionally, when the tail receives a propagated write request, a multicast acknowledgment message can be sent to the multicast group instead of propagating it backwards along the chain. This reduces both the amount of time it takes for a node's object to re-enter the clean state after a write, as well as the client's perceived write delay. Again, no ordering or reliability guarantees are required when multicasting acknowledgments—if a node in the chain does not receive an acknowledgement, it will re-enter the clean state when the next read operation requires it to query the tail.

5 Management and Implementation

Our prototype implementation of Chain Replication and CRAQ is written in approximately 3,000 lines of C++ using the Tame extensions [31] to the SFS asynchronous I/O and RPC libraries [38]. All network functionality between CRAQ nodes is exposed via Sun RPC interfaces.

5.1 Integrating ZooKeeper

As described in §3.4, CRAQ needs the functionality of a group membership service. We use a ZooKeeper file structure to maintain node list membership within each datacenter. When a client creates a file in ZooKeeper, it can be marked as *ephemeral*. Ephemeral files are automatically deleted if the client that created the file disconnects from ZooKeeper. During initialization, a CRAQ node creates an ephemeral file in `/nodes/dc_name/node_id`, where `dc_name` is the unique name of its datacenter (as specified by an administrator) and `node_id` is a node identifier unique to the node's datacenter. The content of the file contains the node's IP address and port number.

CRAQ nodes can query `/nodes/dc_name` to determine the membership list for its datacenter, but instead of having to periodically check the list for changes, ZooKeeper provides processes with the ability to create a *watch* on a file. A CRAQ node, after creating an ephemeral file to notify other nodes it has joined the system, creates a watch on the children list of `/nodes/dc_name`, thereby guaranteeing that it receives a notification when a node is added or removed.

When a CRAQ node receives a request to create a new chain, a file is created in `/chains/chain_id`, where `chain_id` is a 160-bit unique identifier for the chain. The chain's placement strategy (defined in §3.1) determines the contents of the file, but it only includes this chain configuration information, not the list of a chain's current nodes. Any node participating in the chain will query the chain file and place a watch on it as to be notified if the chain metadata changes.

Although this approach requires that nodes keep track of the CRAQ node list of entire datacenters, we chose this method over the alternative approach in which nodes register their membership for each chain they belong to (*i.e.*, chain metadata explicitly names the chain's current members). We make the assumption that the number of chains will generally be at least an order of magnitude larger than the number of nodes in the system, or that chain dynamism may be significantly greater than nodes joining or leaving the system (recall that CRAQ is designed for managed datacenter, not peer-to-peer, settings). Deployments where the alternate assumptions hold can take the other approach of tracking per-chain memberships explicitly in the coordination service. If necessary, the current approach's scalability can also be improved by having each node track only a subset of datacenter nodes: We can partition node lists into separate directories within `/nodes/dc_name/` according to `node_id` prefixes, with nodes monitoring just their own and nearby prefixes.

It is worth noting that we were able to integrate ZooKeeper's asynchronous API functions into our codebase by building tame-style wrapper functions. This allowed us to *twait* on our ZooKeeper wrapper functions which vastly reduced code complexity.

5.2 Chain Node Functionality

Our *chainnode* program implements most of CRAQ's functionality. Since much of the functionality of Chain Replication and CRAQ is similar, this program operates as either a Chain Replication node or a CRAQ node based on a run-time configuration setting.

Nodes generate a random identifier when joining the system, and the nodes within each datacenter organize themselves into a one-hop DHT [29, 45] using these identifiers. A node's chain predecessor and successor are defined as its predecessor and successor in the DHT ring. Chains are also named by 160-bit identifiers. For a chain C_i, the DHT successor node for C_i is selected as the chain's first node in that datacenter. In turn, this node's S DHT successors complete the datacenter subchain, where S is specified in chain metadata. If this datacenter is the chain's first (resp. last), than this first (resp. last) node is the chain's ultimate head (resp. tail).

All RPC-based communication between nodes, or between nodes and clients, is currently over TCP connections (with Nagle's algorithm turned off). Each node maintains a pool of connected TCP connections with its chain's predecessor, successor, and tail. Requests are pipelined and round-robin'ed across these connections. All objects are currently stored only in memory, although our storage abstraction is well-suited to use an in-process key-value store such as BerkeleyDB [40], which we are in the process of integrating.

For chains that span across multiple datacenters, the last node of one datacenter maintains a connection to the first node of its successor datacenter. Any node that maintains a connection to a node outside of its datacenter must also place a watch on the node list of the external datacenter. Note, though, that when the node list changes in an external datacenter, nodes subscribing to changes will receive notification from their local ZooKeeper instance only, avoiding additional cross-datacenter traffic.

5.3 Handling Memberships Changes

For normal write propagation, CRAQ nodes follow the protocol in §2.3. A second type of propagation, called back-propagation, is sometimes necessary during recovery, however: It helps maintain consistency in response to node additions and failures. For example, if a new node joins CRAQ as the head of an existing chain (given its position in the DHT), the previous head of the chain needs to propagate its state backwards. But the system needs to also be robust to subsequent failures *during* recovery, which can cascade the need for backwards propagation farther down the chain (*e.g.*, if the now-second chain node fails before completing its back-propagation to the now-head). The original Chain Replication paper did not consider such recovery issues, perhaps because it only described a more centrally-controlled and statically-configured version of chain membership, where new nodes are always added to a chain's tail.

Because of these possible failure conditions, when a new node joins the system, the new node receives propagation messages both from its predecessor and back-propagation from its successor in order to ensure its correctness. A new node refuses client read requests for a particular object until it reaches agreement with its successor. In both methods of propagation, nodes may use set reconciliation algorithms to ensure that only needed objects are actually propagated during recovery.

Back-propagation messages always contain a node's full state about an object. This means that rather than just sending the latest version, the latest clean version is sent along with all outstanding (newer) dirty versions. This is necessary to enable new nodes just joining the system to respond to future acknowledgment messages. Forward propagation supports both methods. For normal writes propagating down the chain, only the latest version is sent, but when recovering from failure or adding new nodes, full state objects are transmitted.

Let us now consider the following cases from node *N*'s point of view, where L_C is the length of a chain *C* for which *N* is responsible.

Node Additions. A new node, *A*, is added to the system.

- If *A* is *N*'s successor, *N* propagates all objects in *C* to *A*. If *A* had been in the system before, *N* can perform object set reconciliation first to identity the specified object versions required to reach consistency with the rest of the chain.
- If *A* is *N*'s predecessor:
  - *N* back-propagates all objects in *C* to *A* for which *N* is not the head.
  - *A* takes over as the tail of *C* if *N* was the previous tail.
  - *N* becomes the tail of *C* if *N*'s successor was previously the tail.
  - *A* becomes the new head for *C* if *N* was previously the head and *A*'s identifier falls between *C* and *N*'s identifier in the DHT.
- If *A* is within L_C predecessors of *N*:
  - If *N* was the tail for *C*, it relinquishes tail duties and stops participating in the chain. *N* can now mark its local copies of *C*'s objects as deletable, although it only recovers this space lazily to support faster state reconciliation if it later rejoins the chain *C*.
  - If *N*'s successor was the tail for *C*, *N* assumes tail duties.
- If none of the above hold, no action is necessary.

Node Deletions. A node, *D*, is removed from the system.

- If *D* was *N*'s successor, *N* propagates all objects in *C* to *N*'s new successor (again, minimizing transfer to only unknown, fresh object versions). *N* has to propagate its objects even if that node already belongs to the chain, as *D* could have failed before it propagated outstanding writes.
- If *D* was *N*'s predecessor:
  - *N* back-propagates all needed objects to *N*'s new predecessor for which it is not the head. *N* needs to back-propagate its keys because *D* could have failed before sending an outstanding acknowledgment to its predecessor, or before finishing its own back-propagation.
  - If *D* was the head for *C*, *N* assumes head duties.
  - If *N* was the tail for *C*, it relinquishes tail duties and propagates all objects in *C* to *N*'s new successor.
- If *D* was within L_C predecessors of *N* and *N* was the tail for *C*, *N* relinquishes tail duties and propagates all objects in *C* to *N*'s new successor.
- If none of the above hold, no action is necessary.

6 Evaluation

This section evaluates the performance of our Chain Replication (CR) and CRAQ implementations. At a high level, we are interested in quantifying the read throughput benefits from CRAQ's ability to apportion reads. On the flip side, version queries still need to be dispatched to the tail for dirty objects, so we are also interested in evaluating asymptotic behavior as the workload mixture changes. We also briefly evaluate CRAQ's optimizations for wide-area deployment.

All evaluations were performed on Emulab, a controlled network testbed. Experiments were run using the *pc3000*-type machines, which have 3GHz processors and 2GB of RAM. Nodes were connected on a 100MBit network. For the following tests, unless otherwise specified, we used a chain size of three nodes storing a single object connected together without any added synthetic latency. This setup seeks to better isolate the performance characteristics of single chains. All graphed data points are the median values unless noted; when present, error bars correspond to the 99th percentile values.

To determine maximal read-only throughput in both systems, we first vary the number of clients in Figure 4, which shows the aggregate read throughput for CR and CRAQ. Since CR has to read from a single node, throughput stays constant. CRAQ is able to read from all three nodes in the chain, so CRAQ throughput increases to three times that of CR. Clients in these experiments maintained

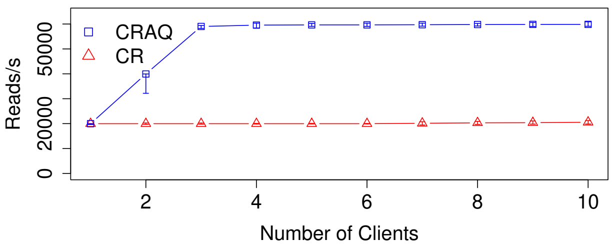

Figure 4: **Read throughput as the number of readers increase: A small number of clients can saturate both CRAQ and CR, although CRAQ's asymptotic behavior scales with chain size, while CR is constant.**

Throughput (in operations/s)				
Type		1st	Median	99th
Read	CR–3	19,590	20,552	21,390
	CRAQ–3	58,998	59,882	60,626
	CRAQ–5	98,919	99,466	100,042
	CRAQ–7	137,390	138,833	139,537
Write	CRAQ–3	5,480	5,514	5,544
	CRAQ–5	4,880	4,999	5,050
	CRAQ–7	4,420	4,538	4,619
Test & Set	CRAQ–3	732	776	877
	CRAQ–5	411	427	495
	CRAQ–7	290	308	341

Figure 5: **Throughput of read and write operations for a 500-byte object and throughput for a test-and-set operation incrementing a 4-byte integer.**

a maximum window of outstanding requests (50), so the system never entered a potential livelock scenario.

Figure 5 shows throughput for read, write, and test-and-set operations. Here, we varied CRAQ chains from three to seven nodes, while maintaining read-only, write-only, and transaction-only workloads. We see that read throughput scaled linearly with the number of chain nodes as expected. Write throughput decreased as chain length increased, but only slightly. Only one test-and-set operation can be outstanding at a time, so throughput is much lower than for writes. Test-and-set throughput also decreases as chain length increases because the latency for a single operation increases with chain length.

To see how CRAQ performs during a mixed read/write workload, we set ten clients to continuously read a 500-byte object from the chain while a single client varied its write rate to the same object. Figure 6 shows the aggregate read throughput as a function of write rate. Note that

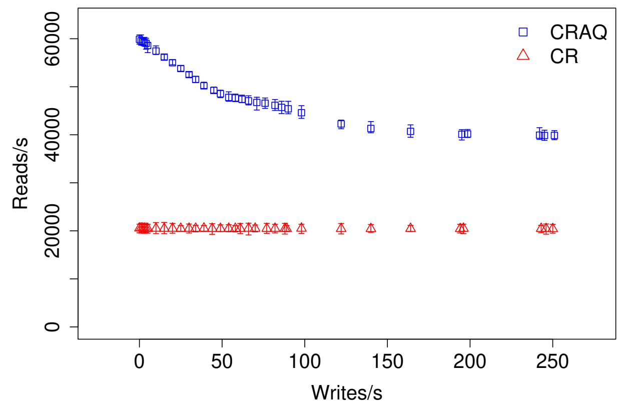

Figure 6: **Read throughput on a length-3 chain as the write rate increases (500B object).**

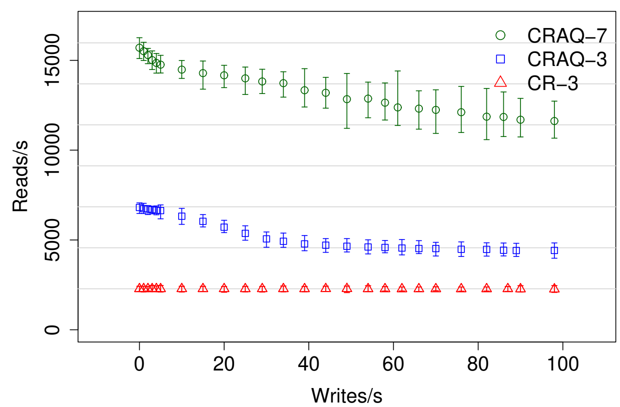

Figure 7: **Read throughput as writes increase (5KB object).**

Chain Replication is not effected by writes, as all read requests are handled by the tail. Although throughput for CRAQ starts out at approximately three times the rate of CR (a median of 59,882 reads/s vs. 20,552 reads/s), as expected, this rate gradually decreases and flattens out to around twice the rate (39,873 reads/s vs. 20,430 reads/s). As writes saturate the chain, non-tail nodes are always dirty, requiring them always to first perform version requests to the tail. CRAQ still enjoys a performance benefit when this happens, however, as the tail's saturation point for its combined read and version requests is still higher than that for read requests alone.

Figure 7 repeats the same experiment, but using a 5 KB object instead of a 500 byte one. This value was chosen as a common size for objects such as small Web images, while 500 bytes might be better suited for smaller database entries (*e.g.*, blog comments, social-network status information, etc.). Again, CRAQ's performance in read-only settings significantly outperforms that of CR with a chain size of three (6,808 vs. 2,275 reads/s), while it preserves good behavior even under high write rates (4,416 vs. 2,259 reads/s). This graph also includes CRAQ performance with seven-node chains. In both scenarios,

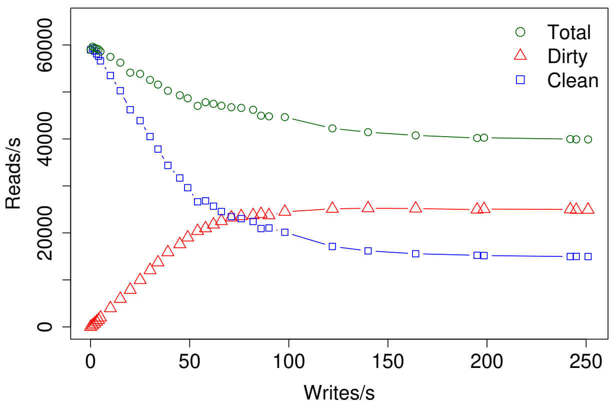

Figure 8: **Number of reads that are dirty vs. clean reads as writes increase (500B key).**

Latency (in ms)						
Type			**Size**	**Med**	**95th**	**99th**
No Load	Reads	Clean	500	0.49	0.74	0.74
			5KB	0.99	1.00	1.23
		Dirty	500	0.98	0.99	0.99
			5KB	1.24	1.49	1.73
	Writes	Length 3	500	2.05	2.29	2.43
			5KB	4.78	5.00	5.05
		Length 6	500	4.51	4.93	5.01
			5KB	9.09	9.79	10.05
Heavy Load	Reads	Clean	500	1.49	2.74	3.24
			5KB	1.99	3.73	4.22
		Dirty	500	2.98	5.48	6.23
			5KB	3.50	6.23	7.23
	Writes	Length 3	500	5.75	7.26	7.88
			5KB	11.61	14.45	15.72
		Length 6	500	20.65	21.66	22.09
			5KB	33.72	42.88	43.61

Figure 9: **CRAQ Latency by load, chain length, object state, and object size within a single datacenter.**

even as the tail becomes saturated with requests, its ability to answer small version queries at a much higher rate than sending larger read replies allows aggregate read throughput to remain significantly higher than in CR.

Figure 8 isolates the mix of dirty and clean reads that comprise Figure 6. As writes increase, the number of clean requests drops to 25.4% of its original value, since only the tail is clean as writes saturate the chain. The tail cannot maintain its own maximal read-only throughput (*i.e.*, 33.3% of the total), as it now also handles version queries from other chain nodes. On the other hand, the number of dirty requests would approach two-thirds of the original clean read rate if total throughput remained constant, but since dirty requests are slower, the number of dirty requests flattens out at 42.3%. These two rates reconstruct the total observed read rate, which converges to 67.7% of read-only throughput during high write contention on the chain.

The table in Figure 9 shows the latency in milliseconds of clean reads, dirty reads, writes to a 3-node chain, and writes to a 6-node chain, all within a single datacenter. Latencies are shown for objects of 500 bytes and 5 KB both when the operation is the only outstanding request (No Load) and when we saturate the CRAQ nodes with many requests (High Load). As expected, latencies are higher under heavy load, and latencies increase with key size. Dirty reads are always slower than clean reads because of the extra round-trip-time incurred, and write latency increases roughly linearly with chain size.

Figure 10 demonstrates CRAQ's ability to recover from failure. We show the loss in read-only throughput over time for chains of lengths 3, 5, and 7. Fifteen seconds into each test, one of the nodes in the chain was killed. After a few seconds, the time it takes for the node to time out and be considered dead by ZooKeeper, a new node joins the chain and throughput resumes to its original value. The horizontal lines drawn on the graph correspond to the maximum throughput for chains of lengths 1 through 7. This helps illustrate that the loss in throughput during the failure is roughly equal to $1/C$, where C is the length of the chain.

To measure the effect of failure on the latency of read and write operations, Figures 11 and 12 show the latency of these operations during the failure of a chain of length three. Clients that receive an error when trying to read an object choose a new random replica to read from, so failures have a low impact on reads. Writes, however, cannot be committed during the period between when a replica fails and when it is removed from the chain due to timeouts. This causes write latency to increase to the time it takes to complete failure detection. We note that this is the same situation as in any other primary/backup replication strategy which requires all live replicas to participate in commits. Additionally, clients can optionally configure a write request to return as soon as the head of the chain accepts and propagates the request down to the chain instead of waiting for it to commit. This reduces latency for clients that don't require strong consistency.

Finally, Figure 13 demonstrates CRAQ's utility in wide-area deployments across datacenters. In this experiment, a chain was constructed over three nodes that each have 80ms of round-trip latency to one another (approximately the round-trip-time between U.S. coastal areas), as controlled using Emulab's synthetic delay. The read client was not local to the chain tail (which otherwise could have just resulted in local-area performance as before).

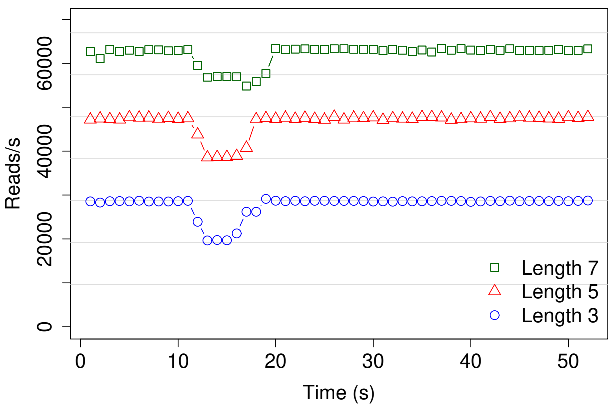

Figure 10: **CRAQ re-establishing normal read throughput after a single node in a chain serving a 500-byte object fails.**

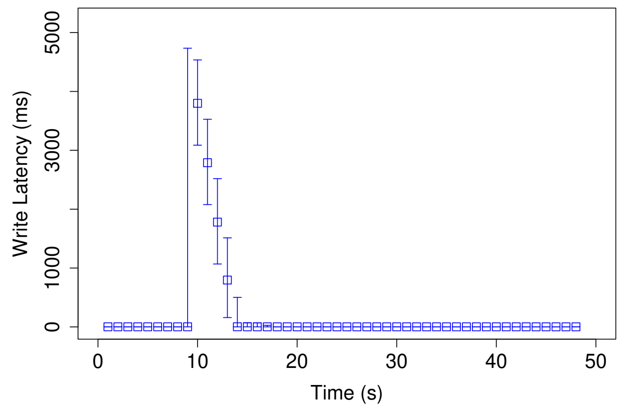

Figure 12: **CRAQ's write latency increases during failure, since the chain cannot commit write operations.**

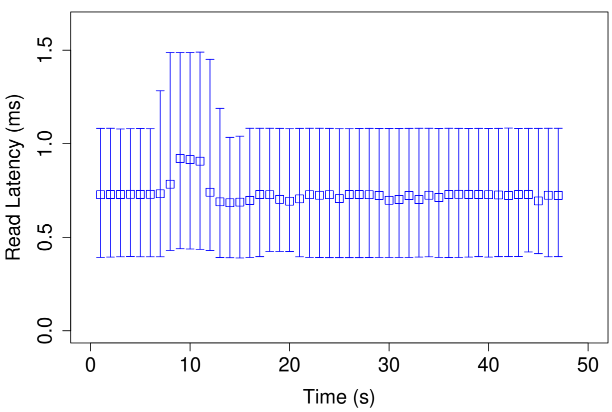

Figure 11: **CRAQ's read latency (shown here under moderate load) goes up slightly during failure, as requests to the failed node need to be retried at a non-faulty node.**

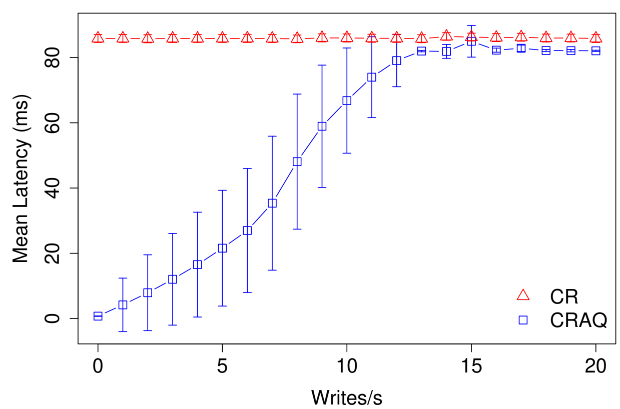

Figure 13: **CR and CRAQ's read latency to a local client when the tail is in a distant datacenter separated by an RTT of 80ms and the write rate of a 500-byte object is varied.**

The figure evaluates read latency as the workload mixture changes; mean latency is now shown with standard deviation as error bars (as opposed to median and 99th percentile elsewhere). Since the tail is not local, CR's latency remains constantly high, as it always incurs a wide-area read request. CRAQ, on the other hand, incurs almost no latency when no writes are occurring, as the read request can be satisfied locally. As the write rate increases, however, CRAQ reads are increasingly dirty, so the average latency rises. Once the write rate reaches about 15 writes/s, the latency involved in propagating write messages down the wide-area chain causes the client's local node to be dirty 100% of the time, leading to a wide-area version query. (CRAQ's maximum latency is ever-so-slightly less than CR given that only metadata is transferred over the wide area, a difference that would only increase with larger objects, especially in slow-start scenarios.) Although this convergence to a 100% dirty state occurs at a much lower write rate than before, we note that careful chain placement allows any clients in the tail's datacenter to enjoy local-area performance. Further, clients in non-tail datacenters that can be satisfied with a degree of maximum-bounded inconsistency (per §2.4) can also avoid wide-area requests.

7 Related Work

Strong consistency in distributed systems. Strong consistency among distributed servers can be provided through the use of primary/backup storage [3] and two-phase commit protocols [43]. Early work in this area did not provide for availability in the face of failures (*e.g.*, of the transaction manager), which led to the introduction of view change protocols (*e.g.*, through leader consensus [33]) to assist with recovery. There has been a large body of subsequent work in this area; recent examples include both Chain Replication and the ring-based protocol of Guerraoui *et al.* [25], which uses a two-phase write protocol and delays reads during uncommitted writes. Rather than replicate content everywhere, one can explore other trade-offs between overlapping read and write sets in strongly-consistent quorum systems [23, 28]. Agreement protocols have also been extended to malicious settings, both for state machine replication [10, 34] and quorum systems [1, 37]. These protocols provide linearizability across *all* operations to the system. This paper does not consider Byzantine faults—and largely restricts its consideration of operations affecting single objects—although it is interesting future work to extend chain replication to malicious settings.

There have been many examples of distributed filesystems that provide strong consistency guarantees, such as the early primary/backup-based Harp filesystem [35]. More recently, Boxwood [36] explores exporting various higher-layer data abstractions, such as a B-tree, while offering strict consistency. Sinfonia [2] provides lightweight "mini-transactions" to allow for atomic updates to exposed memory regions in storage nodes, an optimized two-phase commit protocol well-suited for settings with low write contention. CRAQ's use of optimistic locking for multi-chain multi-object updates was heavily influenced by Sinfonia.

CRAQ and Chain Replication [47] are both examples of object-based storage systems that expose whole-object writes (updates) and expose a flat object namespace. This interface is similar to that provided by key-value databases [40], treating each object as a row in these databases. As such, CRAQ and Chain Replication focus on strong consistency in the ordering of operations *to each object*, but does not generally describe ordering of operations to different objects. (Our extensions in §4.1 for multi-object updates are an obvious exception.) As such, they can be viewed in light of casual consistency taken to the extreme, where only operations to the same object are causally related. Causal consistency was studied both for optimistic concurrency control in databases [7] and for ordered messaging layers for distributed systems [8]. Yahoo!'s new data hosting service, PNUTs [12], also provides per-object write serialization (which they call per-record timeline consistency). Within a single datacenter, they achieve consistency through a messaging service with totally-ordered delivery; to provide consistency across datacenters, all updates are sent to a local record master, who then delivers updates in committed order to replicas in other datacenters.

The chain self-organization techniques we use are based on those developed by the DHT community [29, 45]. Focusing on peer-to-peer settings, CFS provides a read-only filesystem on top of a DHT [14]; Carbonite explores how to improve reliability while minimizing replica maintenance under transient failures [11]. Strongly-consistent mutable data is considered by OceanStore [32] (using BFT replication at core nodes) and Etna [39] (using Paxos to partition the DHT into smaller replica groups and quorum protocols for consistency). CRAQ's wide-area solution is more datacenter-focused and hence topology-aware than these systems. Coral [20] and Canon [21] both considered hierarchical DHT designs.

Weakening Consistency for Availability. TACT [49] considers the trade-off between consistency and availability, arguing that weaker consistency can be supported when system constraints are not as tight. eBay uses a similar approach: messaging and storage are eventually-consistent while an auction is still far from over, but use strong consistency—even at the cost of availability—right before an auction closes [46].

A number of filesystems and object stores have traded consistency for scalability or operation under partitions. The Google File System (GFS) [22] is a cluster-based object store, similar in setting to CRAQ. However, GFS sacrifices strong consistency: concurrent writes in GFS are not serialized and read operations are not synchronized with writes. Filesystems designed with weaker consistency semantics include Sprite [6], Coda [30], Ficus [27], and Bayou [42], the latter using epidemic protocols to perform data reconciliation. A similar gossip-style anti-entropy protocol is used in Amazon's Dynamo object service [15], to support "always-on" writes and continued operation when partitioned. Facebook's new Cassandra storage system [16] also offers only eventual consistency. The common use of memcached [18] with a relational database does not offer any consistency guarantees and instead relies on correct programmer practice; maintaining even loose cache coherence across multiple datacenters has been problematic [44].

CRAQ's strong consistency protocols do not support writes under partitioned operation, although partitioned chain segments can fall back to read-only operation. This trade-off between consistency, availability, and partition-tolerance was considered by BASE [19] and Brewer's CAP conjecture [9].

8 Conclusions

This paper presented the design and implementation of CRAQ, a successor to the chain replication approach for strong consistency. CRAQ focuses on scaling out read throughput for object storage, especially for read-mostly workloads. It does so by supporting *apportioned queries*: that is, dividing read operations over all nodes of a chain, as opposed to requiring that they all be handled by a single primary node. While seemingly simple, CRAQ demonstrates performance results with significant scalability improvements: proportional to the chain length with little write contention—*i.e.*, 200% higher throughput with three-node chains, 600% with seven-node chains—and, somewhat surprisingly, still noteworthy throughput improvements when object updates are common.

Beyond this basic approach to improving chain replication, this paper focuses on realistic settings and requirements for a chain replication substrate to be useful across a variety of higher-level applications. Along with our continued development of CRAQ for multi-site deployments and multi-object updates, we are working to integrate CRAQ into several other systems we are building that require reliable object storage. These include a DNS service supporting dynamic service migration, rendezvous servers for a peer-assisted CDN [5], and a large-scale virtual world environment. It remains as interesting future work to explore these applications' facilities in using both CRAQ's basic object storage, wide-area optimizations, and higher-level primitives for single-key and multi-object updates.

Acknowledgments

The authors would like to thank Wyatt Lloyd, Muneeb Ali, Siddhartha Sen, and our shepherd Alec Wolman for helpful comments on earlier drafts of this paper. We also thank the Flux Research Group at Utah for providing access to the Emulab testbed. This work was partially funded under NSF NeTS-ANET Grant #0831374.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[3] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proc. Intl. Conference on Software Engineering*, Oct. 1976.

[4] Amazon. S3 Service Level Agreement. `http://aws.amazon.com/s3-sla/`, 2009.

[5] C. Aperjis, M. J. Freedman, and R. Johari. Peer-assisted content distribution with prices. In *Proc. SIGCOMM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, Dec. 2008.

[6] M. Baker and J. Ousterhout. Availability in the Sprite distributed file system. *Operating Systems Review*, 25(2), Apr. 1991.

[7] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proc. Very Large Data Bases (VLDB)*, Oct. 1980.

[8] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), 1993.

[9] E. Brewer. Towards robust distributed systems. Principles of Distributed Computing (PODC) Keynote, July 2000.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[11] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. Networked Systems Design and Implementation (NSDI)*, May 2006.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proc. Very Large Data Bases (VLDB)*, Aug. 2008.

[13] CouchDB. `http://couchdb.apache.org/`, 2009.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lak-shman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[16] Facebook. Cassandra: A structured storage system on a P2P network. `http://code.google.com/p/the-cassandra-project/`, 2009.

[17] Facebook. Infrastructure team. Personal Comm., 2008.

[18] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://www.danga.com/memcached/`, 2009.

[19] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.

[20] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. Networked Systems Design and Implementation (NSDI)*, Mar. 2004.

[21] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in G Major: Designing DHTs with hierarchical structure. In *Proc. Intl. Conference on Distributed Computing Systems (ICDCS)*, Mar. 2004.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[23] D. K. Gifford. Weighted voting for replicated data. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Dec. 1979.

[24] Google. Google Apps Service Level Agreement. `http://www.google.com/apps/intl/en/terms/sla.html`, 2009.

[25] R. Guerraoui, D. Kostic, R. R. Levy, and V. Quéma. A high throughput atomic storage algorithm. In *Proc. Intl. Conference on Distributed Computing Systems (ICDCS)*, June 2007.

[26] D. Hakala. Top 8 datacenter disasters of 2007. *IT Management*, Jan. 28 2008.

[27] J. Heidemann and G. Popek. File system development with stackable layers. *ACM Trans. Computer Systems*, 12(1), Feb. 1994.

[28] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Computer Systems*, 4(1), Feb. 1986.

[29] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. Symposium on the Theory of Computing (STOC)*, May 1997.

[30] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Computer Systems*, 10(3), Feb. 1992.

[31] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proc. USENIX Annual Technical Conference*, June 2007.

[32] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov 2000.

[33] L. Lamport. The part-time parliament. *ACM Trans. Computer Systems*, 16(2), 1998.

[34] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Programming Language Systems*, 4(3), 1982.

[35] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Aug. 1991.

[36] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[37] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. Symposium on the Theory of Computing (STOC)*, May 1997.

[38] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Dec 1999.

[39] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: a fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT, June 2005.

[40] Oracle. BerkeleyDB v4.7, 2009.

[41] C. Patridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, Network Working Group, Nov. 1993.

[42] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, , and A. Demers. Flexible update propagation for weakly consistent replication. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.

[43] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Software Engineering*, 9(3), May 1983.

[44] J. Sobel. Scaling out. Engineering at Facebook blog, Aug. 20 2008.

[45] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Networking*, 11, 2002.

[46] F. Travostino and R. Shoup. eBay's scalability odyssey: Growing and evolving a large ecommerce site. In *Proc. Large-Scale Distributed Systems and Middleware (LADIS)*, Sept. 2008.

[47] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[48] Yahoo! Hadoop Team. Zookeeper. `http://hadoop.apache.org/zookeeper/`, 2009.

[49] H. Yu and A. Vahdat. The cost and limits of availability for replicated services. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.