Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data

Athicha Muthitacharoen   Seth Gilbert   Robert Morris
athicha@lcs.mit.edu   sethg@mit.edu   rtm@lcs.mit.edu
MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, 32 Vassar St, Cambridge, MA 02139
Phone: (617) 253-0004 Fax: (617) 258-8607

Abstract

This paper presents Etna, an algorithm for atomic reads and writes of replicated data stored in a distributed hash table. Etna correctly handles dynamically changing sets of replica hosts, and is optimized for reads, writes, and reconfiguration, in that order.

Etna maintains a series of replica configurations as nodes in the system change, using new sets of replicas from the pool supplied by the distributed hash table system. It uses the Paxos protocol to ensure consensus on the members of each new configuration. For simplicity and performance, Etna serializes all reads and writes through a primary during the lifetime of each configuration. As a result, Etna completes read and write operations in only a single round from the primary.

Experiments in an environment with high network delays show that Etna's read latency is determined by round-trip delay in the underlying network, while write and reconfiguration latency is determined by the transmission time required to send data to each replica. Etna's write latency is about the same as that of a non-atomic replicating DHT, and Etna's read latency is about twice that of a non-atomic DHT due to Etna assembling a quorum for every read.

1 Introduction

Distributed hash tables (DHTs) [23, 19] provides a scalable way to store and retrieve data among a large and dynamic set of participating host nodes. Most existing DHTs provide good support for immutable data. However, DHTs that provide fault-tolerant mutable data typically provide no consistency guarantees. There are an increasing number of applications built on top of DHTs that require stronger consistency. These include systems for messaging [7], sharing read/write files [16], resolving names of web objects [26], maintaining bulletin boards [21], and searching large text databases [24]. All of the cited examples depend on a DHT to store and replicate their data, and all of them either assume mutable DHT storage or could be simplified if consistent mutable data were supported.

Existing work on reconfigurable atomic memory service, RAMBO [13, 8], is suitable for a dynamic set of participants, and could be used to provide consistent mutable data in a DHT. However, RAMBO allows multiple active configurations of replicas at any time. As a result, reads and writes in RAMBO can be costly, since RAMBO has to assemble a quorum in every active configuration.

Forseeing the need for efficient and consistent mutable DHT data, we present Etna, a new algorithm for atomic read/write replicated DHT objects. Etna guarantees atomicity regardless of network behavior; for example, it will not return stale data during network partitions. It maintains one consistent configuration per object. Hence, different objects are replicated on different set of nodes, which makes Etna scalable. Etna uses a succession of configurations to ensure that only the single most up-to-date quorum of replicas can execute operations. Etna handles configuration changes using the Paxos distributed consensus algorithm [11]. Etna is designed for low message complexity in the common case in which reads and writes are more frequent than reconfigurations: both reads and writes involve only a single round of communication.

We have implemented Etna on top of the Chord DHT. Experiments in an environment with high network delays show that Etna's read latency is determined by round-trip delay in the underlying network, while write and reconfiguration latency is determined by the transmission time required to send data to each replica. Etna's write latency is about the same as that of a non-atomic replicating DHT, and Etna's read latency is about twice that of a non-atomic DHT due to Etna assembling a quorum for every read.

This paper contains two primary contributions. First, we introduce the first complete design and implementation of an atomic update algorithm in a complete DHT [1]. Second, we provide experimental results demonstrating the performance of the working implementation.

The rest of this paper includes an overview of related work in Section 2, a description of our system model in Section 3, a summary of existing ideas and their interactions with Etna in Section 4, a description of the Etna algorithm in Section 5, a proof of atomicity in Section 6, a performance analysis in Section 7, and a preliminary evaluation of an implementation in Section **??**.

2 Related Work

A few DHT proposals address the issue of atomic data consistency in the face of dynamic membership. Rodrigues et al. [20] use a small configuration service to maintain and distribute a list of all the non-failed nodes. Since every participant is aware of the complete list of active nodes, it is easy to transfer and replicate data while ensuring consistency. However, maintaining global knowledge may limit the approach to small or relatively static systems. Etna uses the Chord DHT [22] to manage the dynamic environment, and augments the basic service to guarantee robust, mutable data.

There have been many quorum-based atomic read/write algorithms developed for static sets of replica hosts (for example [25, 2]). These algorithms assume that the participants are known in advance, and that the number of failures is bounded by a constant.

Group communication services [9] and other virtually synchronous services [3] support the construction of robust and dynamic systems. These algorithms provide stronger guarantees than are required for mutable data storage, implementing totally-ordered broadcast, which effectively requires consensus to be performed for every operation. As a result, the GCS algorithms work best in a low-latency LAN environment. Also, in most GCS systems, whenever a node joins or leaves, a new "view" (i.e., configuration) is created, leading to a potentially slow reconfiguration. Etna uses some of the reconfiguration techniques developed in the GCS protocols. However the read and write operations in Etna require less communication than the multi-phase protocol required to perform totally-ordered broadcast. Also, the rate of reconfiguration can be significantly reduced: a new configuration need only be created when a number of replicas has failed (and no reconfiguration is necessary as a result of join operations).

Prior approaches for reconfigurable read/write memory [10, 5, 18] require that new quorums include processors from the old quorums, restricting the choice of a new configuration. Some earlier algorithms [15, 6] rely on a single process to initiate all reconfigurations. The RAMBO algorithms [13, 8], on the other hand, allow completely flexible reconfiguration, and Etna takes a similar approach. However, RAMBO focuses on allowing reads and writes to proceed concurrently with reconfiguration, resulting in multiple active configurations. Etna, instead, optimizes read and write performance assuming that reconfigurations are rare, by only allowing one active configuration at any time. Hence, Etna needs to contact only a quorum in the active configuration during reads and writes, while RAMBO may have to assemble quorums from multiple active configurations. As a result, read and write operations in Etna are much more efficient than those in RAMBO.

Recent work have applied quorum-based techniques to dynamic systems. Abraham and Malkhi[1] apply probabilistic quorum techniques to a dynamic de Bruijn network. Naor and Wieder [17] suggest a way to apply a quorum system to a dynamic two-dimensional DHT. Etna could make use of either of these techniques to choose consistent quorums. However, since our primary goal is to provide a complete design and implementation of atomic memory on a DHT, we choose to apply the quorum technique to Chord, which is a widely-deployed DHT that has the dynamic ring topology.

3 System Model

We assume a dynamic, cooperating set of nodes in a partially synchronous environment. Communication links may be arbitrarily slow. However, when making progress guarantees and theoretical performance analysis, we assume that messages are delivered within a bounded time, d. Nodes have access to local clocks (which they use for timeouts), but the clocks are not necessarily synchronized. Nodes can crash (fail-stop), join or leave the system at any time.

4 Background

This section describes two components that Etna uses to implement atomic memory, see Figure 1 for an illustration.

[1] Source available at `http://pdos.lcs.mit.edu/chord`.

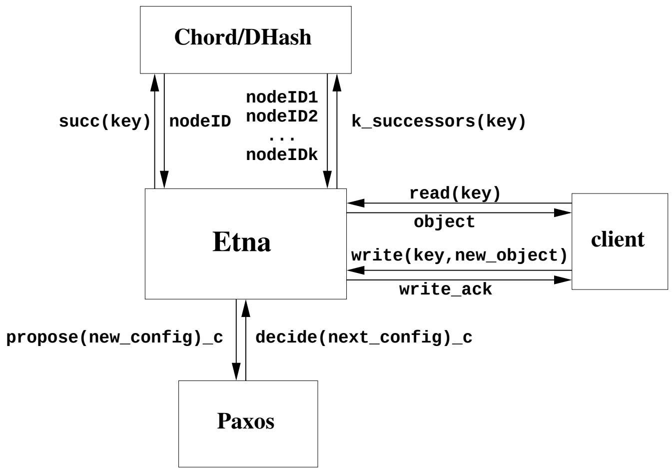

Figure 1: Interaction between Etna and other components.

4.1 Chord

Chord [23, 4] is an efficient, load-balanced DHT. It performs a lookup of an object, given a key, in $O(log(n))$ time, where n is the total number of nodes in the system. Chord arranges that each node knows its current *successor*, the node with the next highest ID; the ID space wraps around at zero. Etna uses Chord to provide it with the node whose ID immediately follows an object key. Etna also uses Chord to provide it with a set of nodes whose IDs are the closest successors of an object key. Etna does not rely on Chord to consistently identify the successor node(s) for an object, since the set of nodes in Chord can rapidly change. Etna maintains a consistent series of replicas for an object regardless of Chord's inconsistency.

4.2 Paxos

Etna uses the Paxos [11, 12] distributed consensus protocol to determine a total-ordering of the replica configurations. We execute a single instance of the Paxos protocol for each configuration; Paxos then outputs the next configuration. All Paxos procedures are subscripted by c, the configuration that is associated with that instance of Paxos.

Paxos is a three-phase commit protocol that implements consensus. As originally described, Paxos consists of two components: an *eventual leader election* service, and an *agreement protocol*. The eventual leader election service ensures that eventually only one node thinks that it is the leader (if the communication network is eventually well-behaved). The agreement protocol allows a node that thinks it is the leader to propose a value. It guarantees that only one value is chosen at most, and that if eventually there is only one leader, then some value is chosen. In the first phase of the protocol, the leader queries for previously proposed decision values; the second phase chooses a decision value.

Etna only makes use of the agreement protocol. Whenever Etna notices that the Chord successor for an object is not the same as the Etna primary, Etna uses Paxos to get a consensus on a new configuration in which they are the same. One or more Etna nodes create proposed configurations and pass them to Paxos:

$$\mathsf{Paxos.propose}(\mathit{proposal\_value})_c$$

When consensus is reached, Paxos calls the Etna

$$\mathsf{decide}(\mathit{decision\_value})_c$$

procedure. At this point, Etna notifies the new configuration of the decision, and the configuration's primary proceeds to locate the object's latest version and serve client requests. Paxos guarantees the following:

Theorem 1 (derived from [11, 12]). *For each instance, c, of the Paxos protocol, if one or more* $\mathsf{decide}(\mathit{decision\_value})_c$ *events occur, then all the values of decision_value are the same.*

If, eventually, a $\mathsf{Paxos.propose}_c$ *occurs at some node i at time t, and no later* $\mathsf{Paxos.propose}_c$ *occurs at any other node j, and node i does not fail, then a* $\mathsf{Paxos.decide}(...)_c$ *occurs by time $t + 2d$, where d is the time it takes for a message to be delivered.*

Since Chord nodes form a dynamic network, its view of the current successor of a object can become inconsistent. It is possible that two nodes think that they are a object's current successor and simultaneously initiate the agreement protocol. This scenario does not violate Etna's correctness, since Paxos guarantees that, given a configuration of replicas, it will always decide on at most one value.

5 The Etna Algorithm

In this section we present Etna, an algorithm that provides fault-tolerant, atomic mutable data in a DHT. For each mutable object, Etna uses Paxos to maintain a consistent series of replica configurations in the face of dynamic membership. Given a configuration of replicas, Etna designates a node as the *primary* and serializes all reads and writes through it.

Per Node State

status	Flag variable, with values `idle`, `active` or `recon_inprog`, initially `idle`
tag	A pair $\langle$*version*, *primaryID*$\rangle$, initially $\langle 0, 0\rangle$
value	Latest object value, initially v_o
new-tag	A pair $\langle$*version*, *primaryID*$\rangle$, initially $\langle 0, 0\rangle$, containing the largest tag of any ongoing write operation
config	Current configuration, with sub-fields *seqnum* $\in \aleph$, initially 0, and *nodes* $\equiv \langle \texttt{node1}, \texttt{node2}, \cdots \rangle$, initially ϕ

Per Operation State

responses	Set of responses for the current operation, initially $\emptyset$

Figure 2: Fields in an Etna object

Network functions

$\texttt{send}_{i,j}(\cdots)$	Sends a message from i to j
$\texttt{recv}_{i,j}(\cdots)$	Receives a message from i to j

Chord functions

`succ()`	Returns the immediate successor of the object's identifier on the Chord ring.
`k_successors()`	Returns the k immediate successors of the object's identifier on the Chord ring.

Etna function

`primary(`c`)`	Returns the primary for a given configuration c

Paxos functions

`propose(`$\cdots$`)`	Proposes a new configuration.

Figure 3: Auxiliary functions, provided by Chord, Etna, Paxos, and the network.

5.1 Object State

Etna provides atomicity for each mutable object, which extends to the entire DHT. By the composability of atomic objects [14], all mutable objects in the DHT form an atomic memory. Therefore, we describe our protocol in terms of a single object.

To provide fault-tolerance, Etna replicates a mutable object at k different nodes, where k is the system-wide replication factor. We call this replica set a *configuration* of nodes responsible for an object. Etna initiates reconfigurations in order to arrange that an object's replicas are the nodes that immediately succeed the object's key in the Chord ID space, and the object's immediate successor node is the *primary* in the configuration. For each object, a replica keeps a *tag*, which is a pair $\langle$*version*, *primaryID*$\rangle$; a *status* flag, which designates the phase of operation to the object; a *new-tag*, which is used during write operations; and a *config* variable, which contains the configuration sequence number and the IDs for the nodes in the object's configuration. Etna increments the sequence number for each new configuration. Figure 2 summarizes the fields in a mutable object. We refer to a mutable object as simply an object throughout the rest of the paper.

5.2 Inserting a New Object

To insert a new object, the writer passes the object's data to the Etna client on the local machine. Etna extends the object with the fields in Figure 2. Because the object is new, Etna can directly insert it at the initial replicas, set to be the k immediate successors to the object's ID.

5.3 Read Protocol

To read an object with key *bID*, the reader sends a `read` RPC to the node i which is the immediate successor of *bID*. i checks if it believes it is the current *primary* of *bID* and if *bID* is not going through a reconfiguration (*status* = `active`). If both conditions are true, it sends a `GET` RPC to each node in *config*. If not, the replicas may be going through membership reconfiguration, so i returns an error. The reader will retry periodically.

When a node j receives a `GET` RPC for *bID*, it looks in *config* to see if the sender is the current primary of *bID* and if *status* = `active`. If both conditions are true, j returns with a positive ack. If not, it returns an error.

If i collects more than $k/2$ positive acks, it returns its own stored copy of the object value to the reader. If i fails to assemble a majority after a certain time, it returns an error. Figure 4 shows the pseudocode for the read protocol.

5.4 Write Protocol

To write object *bID*, the writer sends a `write` RPC to the successor of *bID*, node i. i consults its local state to verify that it believes it is the current primary of *bID* and that *status* is `active`. If both conditions are true, i starts the write protocol:

1. Node i assigns a new version number to this write, giving it tag $\langle$*new-tag.version* $+ 1, i\rangle$.
2. Node i sends a `put` RPC to each replica node in *config.nodes*, including the write's tag and value.
3. When a node j receives a `put` RPC for *bID*, it ignores the RPC if *status* is not `active` or if the sender isn't the primary. If the write's tag is higher than the stored tag, j updates its stored object. This ensures that a replica is not confused if concurrent writes arrive from the primary out of order. Node j then returns an ack to the sender.
4. When node i receives positive acks from a majority of the replicas, it updates its own copy of the

Read protocol for the primary:

```
Procedure recv(read)_{c,i}
  if i = primary(config) then
    if status = active then
      responses ← ∅
      ∀j ∈ config.nodes do
        send(get, c, config.seqnum)_{i,j}

Procedure recv(get-ack, c, seqnum)_{j,i}
  if seqnum = config.seqnum then
    responses ← responses ∪ {j}
  if | responses | > ⌈ k/2 ⌉ then
    if status = active then
      send(tag, value)_{i,c}
```

Read protocol for the replicas (including the primary):

```
Procedure recv(get, c, seqnum)_{j,i}
  if status = active then
    if (seqnum = config.seqnum) then
      send(get-ack, c)_{j,i}
```

Figure 4: Pseudo-code for the read protocol.

Write protocol for the primary:

```
Procedure recv(write, c, new-val)_{c,i}
  if i = primary(config) then
    if status = active then
      new-tag ← ⟨new-tag.version + 1, i⟩
      op-object ← ⟨new-tag, new-val⟩
      responses ← ∅
      ∀j ∈ config.nodes do
        send(put, c, config.seqnum, op-object)_{i,j}

Procedure recv(put-ack, c, op-object, seqnum)_{j,i}
  if seqnum = config.seqnum then
    responses ← responses ∪ {j}
  if | responses | > ⌈ k/2 ⌉ then
    if status = active then
      if op-object.tag > tag then
        tag ← op-object.tag
        value ← op-object.value
      send(put-ack)_{i,c}
```

Write protocol for the replicas (including the primary):

```
Procedure recv(put, c, seqnum, op-object)_{j,i}
  if status = active and i ≠ j then
    if (seqnum = config.seqnum) then
      if op-object.tag > tag then
        tag ← op-object.tag
        value ← op-object.value
      send(put-ack, c, op-object, config.seqnum)_{j,i}
```

Figure 5: Pseudo-code for the write protocol.

object, though only if it has completed no subsequent concurrent write. Node i then replies to the client. If i fails to assemble a majority after a certain time, it returns an error.

Figure 5 illustrates the write protocol. The primary assigns increasing version numbers to writes in the order that they arrive at the primary, but issues the writes to the replicas in parallel.

5.5 Reconfiguration Protocol

Etna must change the configuration of nodes responsible for an object when a replica leaves (to maintain the k replication factor) and when a new node joins that would be the object's successor (so that the primary is the Chord successor and is thus easy to find). Etna maintains only one configuration at a time, rather than multiple configurations as in Rambo [13]; this allows Etna to be simpler and have higher performance in all but the highest-churn environments. Etna uses the Paxos [11] distributed consensus protocol to decide on the next configuration.

If an Etna node i notices that the set of Chord successors for a object *bID* does not match the set of replicas in the object's *config*, it tries to initiate a reconfiguration:

Case I If i notices that it is the immediate successor of *bID*, it collects some information that will serve as a configuration proposal for a Paxos execution. The proposal has the form $\mathit{proposal\_value} \equiv \langle \mathit{new\_config}, \mathit{object\_copy} \rangle$. i sets *new_config* to be the k immediate successors of *bID*. i sends a `recon-get` RPC to each node in *config* asking for its current object value and tag, waits for a majority of replicas to respond, and uses the most up-to-date response as the *object_copy* in the proposal.

When a replica receives a `recon-get` RPC, it sets *status* to `recon_inprog` and stops processing all reads and writes.

When i has assembled a majority of `recon-get` responses, it uses Paxos to propose its *proposal_value* to the nodes in the old configuration. Paxos calls the `decide` function at each node that proposed a new configuration, with the consensus configuration information; the proposer(s) send the new configuration and most up to date object to the replicas in the new configuration.

Case II If i is not immediate successor of *bID*, i sends

```
Reconfiguration protocol for the primary:

   Procedure recon()_i
2    if config.nodes ≠ k_successors() then
       if i = succ() then
4        new-config ← k_successors()
         status ← recon_inprog
6        proposed ← false
         responses ← ∅
8        ∀j ∈ config.nodes
           send(recon-get)_{i,j}
10
   Procedure recv(recon-ack, new-tag, new-value, seqnum)_{i,j}
12   if seqnum = config.seqnum then
       if new-tag > tag then
14       tag ← new-tag
         value ← new-value
16     responses ← responses ∪ {j}
     if | responses | > ⌈ k/2 ⌉ and proposed ≠ true then
18     proposed ← true
       object-copy ← ⟨tag, value⟩
20     Paxos.propose(new-config, object-copy)_config

22 Procedure decide(new-config, object-copy)_c
     ∀j ∈ new-config.nodes
24     send(update, new-config, object-copy)_{i,j}

Reconfiguration protocol for the replicas:

   Procedure recv(recon-get, seqnum)_{j,i}
25   if (seqnum = config.seqnum) then
       status ← recon_inprog
27     send(recon-ack, tag, value, config.seqnum)

29 Procedure recv(update, new-config, object-copy)_{j,i}
     if i ∈ new-config.nodes then
31     if new-config.seqnum > config.seqnum then
         status ← active
33       config ← new-config
         if object-copy.tag > tag then
35         tag ← object-copy.tag
           value ← object-copy.value
```

Figure 6: Pseudo-code for reconfiguration.

the *tag*, *value*, and *config* information to the immediate successor of *bID*, asking the successor to initiate a reconfiguration.

Figure 6 shows the pseudo-code for the reconfiguration protocol. During reconfiguration, nodes in the current configuration become inactive (stops serving write and read requests). If reconfiguration fails, there will be no active configuration. Section 7 discusses liveness.

6 Atomicity

In this section, we show that Etna correctly implements an atomic read/write object. Throughout this section, we consider only a single object, b. Since atomic memory is composable, this is sufficient to show that Etna guarantees atomic consisteny. We omit references to b for the rest of this section.

We use the partial-ordering technique, described in Lynch [14]. We rely on the following lemma:

Lemma 1 (Lemmas 13.16 and 13.10 in [14]). *Let α be any well-formed, finite execution of algorithm X (implementing a read/write atomic object) in which every operation completes. Let Π be the set of all operations in α.*

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties:

1. *For any operation $A \in \Pi$, there are only finitely many operations $B \in \Pi$ such that $B \prec A$.*
2. *If A finishes before B starts, then it cannot be the case that $B \prec A$.*
3. *If A is a write operation in Π and B is any operation in Π, then either $A \prec B$ or $B \prec A$.*
4. *The value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or v_o, if there is no such write).*

Then every well-formed execution of algorithm X satisfies the atomicity property.

We consider an arbitrary well-formed execution, α, of the Etna algorithm in which every read and write operation completes. (Lemma 13.10 in Lynch [14] indicates that it is sufficient to consider only such executions.) We first define a partial order on the read and write operations in α, and then show that this partial order has the properties required by Lemma 1. Finally, we conclude that Etna guarantees atomic consistency.

Partial Order. We first order the read and write operations in α based on their tags. For a read or write operation $A \in \alpha$, initiated at node i, we define $tag(A) = tag_i$ immediately before the operation returns to the reader or writer; that is, $tag(A)$ is the value of the object's tag when i sends the result back to the client (Figure 5, Line 18 and Figure 4, Line 13).

For a reconfiguration operation, π, we define $tag(\pi) = object\text{-}copy.tag_i$ immediately before the call to Paxos.propose (Figure 6, Line 20). We then define the partial order $\prec$: (i) For any two operations A and B in α: if $tag(A) < tag(B)$, then $A \prec B$. (ii) For any write operation A in α, and any read operation B in α: if $tag(A) = tag(B)$ then $A \prec B$. We show in Theorem 2 that it is straightforward to see that this partial-order satisfies Properties 1, 3, and 4 of Lemma 1. The primary goal of the rest of this section is to show that this ordering satisfies Property 2.

Atomicity Proof. Our first goal is to show that when a new configuration is installed, no information is lost. Let configuration c_0 be the initial configuration, and configuration $c_{\ell+1}$ be the unique configuration decided on by $\mathsf{Paxos}_{\ell,c_\ell}$. (Theorem 1 ensures that this is, in fact, unique.) If $\mathsf{Paxos}_{\ell,c_\ell}$ does not terminate, then c_ℓ is undefined.

Recall that when Paxos is initiated, the process performing the reconfiguration includes *object-copy*, the latest copy of the object, in the proposal. That is, $\mathsf{Paxos}_{\ell,c_\ell}$ is initiated with at least one call to:
$\mathsf{Paxos.propose}(\textit{new\_config}, \textit{object\_copy})_{\ell,c_\ell}$.
Define $\textit{tag}(c_0)$ to be $\langle 0, 0\rangle$ and $\textit{tag}(c_{\ell+1})$ to be *object_copy.tag* of the successful proposal to $\mathsf{Paxos}_{\ell,c_\ell}$. We sometimes refer to $\textit{tag}(c_{\ell+1})$ as the *initial tag* of configuration $c_{\ell+1}$. We want to show that the initial tags of the configurations are nondecreasing:

Lemma 2. *Let c_ℓ and $c_{\ell+1}$ be configurations installed in α. Then $\textit{tag}(c_\ell) \leq \textit{tag}(c_{\ell+1})$.*

Proof. No replica in configuration c_ℓ can send any response until it has received an update message (Figure 6, Lines 29– 36), which causes the replica to set its *status* to active. Therefore every response to the recon-get message (Figure 6, Lines 24– 27) during the recon that proposes configuration $c_{\ell+1}$ must include a tag no smaller than $\textit{tag}(c_\ell)$. Therefore $\textit{tag}(c_\ell) \leq$ *object-copy.tag* in the proposal from $c_{\ell+1}$, from which the result follows. □

It then follows immediately by induction:

Corollary 1. *If c_ℓ and c_k are two configurations in α, and $\ell < k$, then $\textit{tag}(c_\ell) \leq \textit{tag}(c_k)$.*

We next consider a read or write operation that occurs in configuration c_ℓ (for some $\ell \geq 0$). We want to show that if A is an operation that completes in c_ℓ, then the value A returns has a tag no smaller than $\textit{tag}(c_\ell)$.

For read or write operation A, let $\textit{conf}(A) = \textit{config.seqnum}$ when i sends the result back to the client (Figure 5, Line 18 and Figure 4, Line 13).

Lemma 3. *Let B be a read or write operation in α, and assume that $\textit{conf}(B) = \ell$. Then $\textit{tag}(c_\ell) \leq \textit{tag}(B)$, and if B is a write operation then the inequality is strict.*

Proof. The reconfiguration to install configuration c_ℓ concludes when a primary, i, wins a Paxos decision (Figure 6, Lines 22– 24) and sends messages to the new replicas. Notice that the decision includes the *object-copy* determined when the reconfiguration began. Therefore, by the time the configuration is installed, $\textit{tag}(c_\ell) \leq \textit{tag}_i$. As a result, every operation initiated at node i has a tag greater than or equal to $\textit{tag}(c_\ell)$ and with write operations the inequality is strict, since write increments the tag. □

Next, we relate the tag of a read or write operation to the tag of the *next* configuration. We want to show that if a read or write operation completes, the information is transferred to the next configuration.

Lemma 4. *Let A be a read or write operation in α, and assume that $\textit{conf}(A) = \ell$. If configuration $c_{\ell+1}$ is installed in α, then $\textit{tag}(A) \leq \textit{tag}(c_{\ell+1})$.*

Proof. First, notice that the *value* of the primary always reflects a write operation that has updated a majority of the replicas. Therefore if A is a read operation, there is a write operation, A', that wrote the tag and value returned by A to a majority of the replicas. If A is a write operation, define $A' = A$.

Since operation A' completes in configuration c_ℓ, there exists a set of at least $\lceil k/2 \rceil$ nodes in configuration c_ℓ that a send a response for A' to the primary. Call this set of nodes W (for "writers").

Since configuration $c_{\ell+1}$ is installed in α, there exists a set of at least $\lceil k/2 \rceil$ nodes in configuration c_ℓ that send a response to the recon-get message during the reconfiguration. Call this set of nodes R (for "readers").

Notice that since there are k nodes in configuration c_ℓ, and both R and W contain at least $k/2$ nodes, there is at least one node, j, in both R and W. Node j sends a response both for operation A' and for the successful reconfiguration resulting in $c_{\ell+1}$.

We claim that node j sends the response for operation A' before the response for recon-get. As soon as node j sends a response for a recon-get, it sets its $\textit{status}_j$ to recon-in-progress, at which point it ceases responding to requests. Since we know that j sends a response to operation A', it must send this response prior to the first time it receives a recon-get request.

We conclude, then, that the primary sends its put request for A' to replica j prior to j sending its response to the recon-get request. Therefore, $\textit{tag}(A) = \textit{tag}(A') \leq \textit{tag}(c_{\ell+1})$. □

In the final preliminary lemma, we show that the configurations used by operations are non-decreasing. That is, if operation A occurs in one configuration, then a later operation B cannot occur in an earlier configuration.

Lemma 5 (sketch). *Let A and B be two read or write operations in α. Assume that operation A completes before operation B begins. Then $conf(A) \leq conf(B)$.*

Proof. If A completes in configuration c_ℓ, then some reconfiguration completes prior to A for c_ℓ. During that reconfiguration, a majority of replicas in configuration $c_{\ell-1}$ were sent a recon-get message notifying them to cease processing read and write requests. By induction, a majority of replicas from *all* earlier configurations received such messages. Therefore operation B can not complete *after* A using an earlier configuration. □

Finally, we relate read and write operations.

Lemma 6. *Let A and B be two read and write operations in α where A completes before B begins. Then $tag(A) \leq tag(B)$, and if B is a write operation then the inequality is strict.*

Proof. We break the proof down into two cases: (i) A and B complete in the same configuration, and (ii) A completes in an earlier configuration than B. Lemma 5 shows that A cannot complete in a later configuration than B.

First, assume that $k = conf(A) = conf(B)$. Let node i be the primary of configuration c_k. Both operations originate at node i. Therefore, when operation B begins, the tag of i is at least as large as $tag(A)$. If B is a write operation, then i increments the tag, and the inequality is strict.

Next, consider the case where $conf(A) < conf(B)$. Notice that $tag(A) \leq tag(conf(A))$, by Lemma 4. Next, notice that $tag(conf(A)) \leq tag(conf(B))$, by Corollary 1. Third, notice that $tag(conf(B)) \leq tag(B)$, and if B is a write operation, the inequality is strict, by Lemma 3. Combining the inequalities implies the desired result. □

Finally, we prove the main theorem:

Theorem 2. *The Etna algorithm correctly implements an atomic read/write object.*

Proof. We show that the protocol satisfies the four conditions of Lemma 1. For an arbitrary execution α in which every read and write operation completes, we demonstrate that the partial-ordering, $\prec$, satisfies Properties 1–4 of Lemma 1.

1. Immediate.
2. It follows from Lemma 6 that if A completes before B begins, then $tag(A) \leq tag(B)$. Therefore $B \not\prec A$.
3. If A and B are write operations, then it follows immediately that $tag(A) \neq tag(B)$, since the tags are unique (as they consist of a sequence number and a node identifier to break ties).
   If A is a write operation and B is a read operation and $tag(A) = tag(B)$ then $A \prec B$. Otherwise, if $tag(A) \neq tag(B)$, then either $A \prec B$ or $B \prec A$, depending on whether A or B has a larger tag.
4. This follows by the definition of the partial order. If B is a read operation, then $tag(B)$ is the tag of the write operation, A, whose value B returns, or the initial tag. Therefore, either A is the last preceding write operation (since $tag(A) = tag(B)$ or B returns v_0.

□

7 Theoretical Performance

As in all quorum based algorithm, the performance of the algorithm depends on enough replicas remaining alive. We assume that if a node crashes, the remaining live nodes in the relevant configurations notice the crash and reconfigure quickly enough to maintain a live majority. If a majority of the nodes in a configuration fail, then operations can no longer complete. A recovery protocol could attempt to collect the values from the remaining replicas, at the expense of atomicity; we do not address recovery from failed configurations in this paper.

During intervals in which the primary does not fail, the algorithm is efficient. A write operation requires a single round of communication to propagate the new value to a quorum. A read operation also requires only a single round of communication, involving only small control messages, since the primary supplies the data. For the purpose of this section, we assume that each message is delivered in time d:

Lemma 7. *If a read or write operation begins at time t, and the primary does not fail by time $t + 2d$, then the operation completes by time $t + 2d$.*

A reconfiguration is somewhat more expensive, requiring three and a half rounds of communication. As soon as a new primary is established, it queries the old replicas for the latest value of the block. It then begins

Paxos, which requires two rounds of communication to arrive at a decision. Finally, it updates the new configuration. In our implementation, we piggy-back the `recon-get` RPC with the first RPC of Paxos, reducing the communication to two and a half rounds. The following lemma reflects this optimization.

Lemma 8. *If node i is designated the primary at time t, and i does not fail by time $t + 5d$, then the new configuration is installed by time $t + 5d$.*

Recall that when a reconfiguration takes place, ongoing read and write operations may fail to complete. We assume that, in this case, the client retries the operation at the new primary. Combining the two previous lemmas, we see that even if a primary fails, a read or write operation completes within $7d$ after a new primary is designated.

8 Experimental Evaluation

We are in the process of conducting more extensive experiments.

9 Conclusion

This paper describes Etna, an algorithm for atomic mutable blocks in a distributed hash table. Etna correctly handles a dynamically changing set of replica hosts, using protocols optimized for situations in which reads are more common than writes or replica set changes. Etna uses Paxos to agree on a sequence of replica configurations, and only allows operations when a majority of replicas from the active configuration are available. Etna's write latency is comparable to that of non-atomic replicated DHTs, and its read latency is approximately twice that of a DHT.

References

[1] Ittai Abraham and Dahlia Malkhi. Probabilistic quorums for dynamic systems. In *Proceedings of the 17th Annual Conference on Distributed Computing*, 2003.

[2] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[3] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, December 1987.

[4] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[5] Danny Dolev, Idit Keidar, and Esti Yeger Lotem. Dynamic voting for consistent primary components. In *Proc. of the Sixteenth Annual ACM Symp. on Principles of Distributed Computing*, pages 63–71. ACM Press, 1997.

[6] B. Englert and Alex A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of the International Conference on Distributed Computer Systems*, pages 454–463, 2000.

[7] Alan Mislove et. al. POST: A secure, resilient, cooperative messaging system. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[8] Seth Gilbert, Nancy A. Lynch, and Alex A. Shvartsman. RAMBO II:: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. of the Intl. Conference on Dependable Systems and Networks*, pages 259–269, June 2003.

[9] *Communications of the ACM, Special section on Group Communication Systems*, volume 39(4), 1996.

[10] S. Jajodia and David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *Trans. on Database Systems*, 15(2):230–280, 1990.

[11] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[12] Leslie Lamport. Paxos Made Simple. 2001.

[13] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service. In *Proceedings of the 16th International Symposium on DIStributed Computing (DISC '02)*, Toulouse, France, October 2002.

[14] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[15] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Twenty-Seventh Annual Intl. Symposium on Fault-Tolerant Computing*, pages 272–281, June 1997.

[16] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[17] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. In *Proceedings of the 22nd Symposium on Principles of Distributed Computing*, 2003.

[18] Roberto De Prisco, Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. A dynamic primary configuration group communication service. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 64–78, September 1999.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, August 2001.

[20] Rodrigo Rodrigues, Barbara Liskov, and Liuba Shrira. The design of a robust peer-to-peer system. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.

[21] E. Sit, F. Dabek, and J. Robertson. UsenetDHT: A low overhead Usenet server. In *Proc. of the Third International Workshop on Peer-to-Peer Systems*, February 2004.

[22] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, August 2001.

[23] I. Stoica, R. Morris, David Liben-Nowell, D. Karger, M. Frans Kaashoek, Frank Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Transactions on Networking*.

[24] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. ACM SIGCOMM Conference*, August 2003.

[25] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, 1987.

[26] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from DNS. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.