Key Consistency in DHTs

Sriram Sankararaman
Byung-Gon Chun
Chawathe Yatin
Scott Shenker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-21
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-21.html

November 29, 2005

University of California, Berkeley. Technical Report UCB/EECS. November 2005

Key Consistency in DHTs

Sriram Sankararaman[†], Byung-Gon Chun[†], Yatin Chawathe, and Scott Shenker[†]
[†]University of California, Berkeley
{sriram_s,bgchun,shenker}@cs.berkeley.edu, yatin.research@chawathe.com

Abstract

Most DHTs are designed more for scalability than for consistency, and thus do not provide strong guarantees on the consistency of data. In this paper, we focus on key consistency rather than data consistency: key consistency requires that no key be owned by more than one root. We briefly show how key consistency can be used to support atomic DHT operations and then propose a mechanism to achieve key-consistency. We have tested our algorithm through simulation and a Planetlab deployment, and find that it provides high availability in the face of node churn and packet drops.

1 Introduction

Distributed Hash Tables (DHTs) abstract away the details of distributed storage and retrieval, providing application developers with a simple and efficient *put*/*get* interface. To support this interface, most DHTs use a form of consistent hashing where each node in the DHT serves as the *root* for a range of the key space, serving the *put*/*get* requests for those keys.[1]

Consistency is an important but elusive goal in DHTs. While there have been some attempts to provide data consistency guarantees in DHTs, they typically involve significant overhead and complicated designs [6–8]. Most DHT designs put more emphasis on scalability than consistency and, to achieve scalability, provide no guarantee that each *get* will return the latest data. As a result, network partitions, churn and replication can all cause a DHT to return values that have been removed or overwritten, or to fail to find any value at all.

[1]In what follows, we will only consider DHT routing algorithms that only terminate at a root (unless it can't find one) rather than a replica.

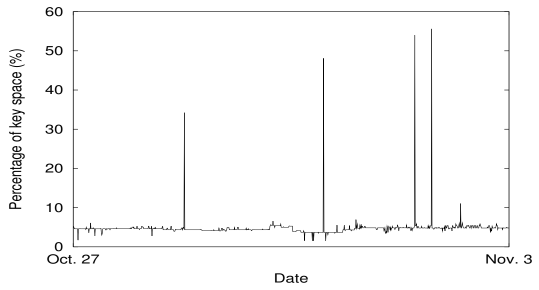

Figure 1: Percentage of the keyspace that is inconsistent over a 7-day time period in a PlanetLab deployment of OpenDHT code.

In this paper we choose to focus on key consistency rather than data consistency. We call a DHT *key-consistent* for a key k if no more than one node claims to be the root for k. This seems to be a rather modest goal, but in practice it is often violated. Figure 1 shows the inconsistent fraction of the keyspace in a PlanetLab deployment of the OpenDHT code [9]. For most of the measured period, around 5% of the keys have multiple roots (presumably due to nontransitive routing [4]), and at certain times (presumably due to increased churn), the fraction of inconsistent keys spikes to a much higher value.

The figures 2 (a) and (b) show scenarios in which a key has multiple roots. In 2 (a), nodes A and C cannot talk to B. A considers C to be its successor. Similarly, D cannot talk to A but can talk to B. A lookup for key k that reaches D is forwarded to B while one that reaches A is forwarded to C. Figure 2 (b), node B that was disconnected, rejoins with an identifier between A and C. A does not know of the presence of B. Lookups on key k that reach are forwarded to C. If B were to have fingers pointing to it, then it would be forwarded lookups for key k as well. These inconsistent configurations makes it dif-

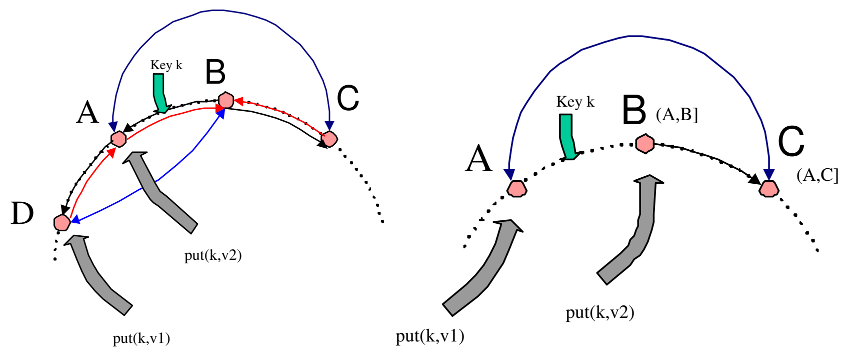

Figure 2: Instances of multiple root in the chord DHT.(a) The effect of transitive links. (b) The effect of churn

ficult to modify the values stored in the DHT.

Our goal is to develop a lightweight technique to achieve key consistency. The mechanism we propose designates nodes as *authorized* roots in such a way that there is never more than one authorized root for a particular key. Our mechanism is simple, lightweight, and scalable, and makes no assumptions about message delivery times, system partitions, or time synchronization across nodes (although it does assume clocks progress at the same speed).[2] Moreover, the consistency mechanism is completely separate from the normal DHT routing and replication algorithms, so that it does not in any way impair the performance of the DHT.

Why do we care about key consistency when it does not guarantee data consistency? First, key-inconsistency is an important cause of data inconsistency, because if several nodes claim ownership of a particular key k, a *get* on a key k may reach a different node from the one contacted by a previous *put* and therefore return inconsistent results. Thus, reducing key inconsistency can reduce data inconsistency, even if it does not provide strict data consistency guarantees. Second, if the DHT is key-consistent, all operations can be serialized through the single authorized root. This serialization can be used to build atomic DHT primitives. Thus, the presence of key consistency will allow us to build data primitives with better semantics, even if they fall short of full data consistency.

To put this in context, consider the spectrum of application consistency requirements. On one end of the spectrum, applications such as file sharing can tolerate the level of data inconsistency that current DHTs provide and do not need any additional data consistency guarantees. At the other end of the spectrum, applications like distributed file systems require long-term data consistency where each *get* produces the latest *put* data, no matter how long ago that put occurred. As noted above, algorithms to achieve this level of consistency are quite complex, and involve complicated agreement algorithms among the various replicas of the data (see, for example, [7]).

Intermediate between these extremes are applications that require atomicity rather than long-term consistency. For example, Place Lab [2] uses DHTs to build a tree-based index data structure for range queries and requires all tree updates to be performed atomically so as to prevent corruption of the data structure. It is to this category of applications that key consistency delivers significant value.

To clarify our goal, we note that there are roughly two main causes of data inconsistency: loss and confusion. Data loss occurs when a root fails (and, if the data is replicated, a sufficient numbers of replicas also fail), taking with it the most recent put. Confusion occurs when more than one node thinks it is the root. Key consistency addresses the second problem but not the first. As we describe in the next section, we show that this is sufficient to build some atomic operations that are not achievable in traditional DHTs.

2 Atomic Mutable Data

In this section, we show how a key-consistent DHT can be changed to support atomic updates. We first describe the role of key consistency, and then describe the modifications to the *put*/*get* operations.

[2]Although our algorithm could be adapted to handle small differentials in clock rates.

The key consistency mechanism allows the DHT to support an *auth* bit. A $put(k)$ or $get(k)$ operation has its *auth* bit set to true if and only if the operation reaches the authorized root for k. The key consistency mechanism also allows the DHT to track the *chain of custody*. Let $history(k)$ indicate the longest contiguous window of time into the past for which the data under a key has always been under the custody of an authorized root for that key. When a new node n joins the DHT, the current root m for a key k that belongs to n transfers data stored under k to n. m can also delegate authority over k to n if it itself was the authorized root for k so that key-consistency is still satisfied. Such a handoff of key k is called "clean". The value of $history(k)$ stores the lifetime of k across "clean" handoffs. Now suppose n crashes and m again becomes authorized for k in the next round of authorization algorithm, $history(k)$ would be set to 0 because the handoff was dirty (the value under k is treated as new data). Thus, the value of $history(k)$ indicates the duration of the clean chain of custody of key k.

To discuss atomic operations, consider a read-modify-write instance. A node retrieves the values stored under key k using $get(k)$, modifies these values, and writes them back using $put(k,v)$. We modify the *get* operation so that a $get(k)$ returns the value stored under key k, a version number $version(k)$ for k and a bit $auth(k)$. The version number is incremented when the key is updated. The client retries the *get* if $auth(k)$ is 0. We define an *atomic_put* operation : $atomic\_put(k, value, version, t)$. *version* is the version number returned with the corresponding *get* while t is the time elapsed between the *get* and the *atomic_put*. The *atomic_put* succeeds only if the root is authorized for key k, the version number of k is unchanged and $t < history(k)$. The successful *atomic_put* increments the version number of the key. When multiple concurrent updates are attempted, all the *atomic_put*s reach the same authorized root and only one of these wins the race.

This atomicity of this operation relies on key consistency, and the rest of this paper is devoted to the description and evaluation of our proposed mechanism for achieving key consistency.

3 Authorization Algorithm

We would like the authorization algorithm to ensure that the DHT reaches and then remains in a key-consistent state. The authorization algorithm ensures the following properties. The proofs of the theorems can be found in Section 3.4.

Theorem 1 *In the DHT D, if no more than a single node is authorized for a key at time t, the key will be authorized to at most one node for all $t' > t$.*

Theorem 2 *Consider a DHT D in an arbitrary state at time t, i.e., more than one node may be authorized for a key. An authorization algorithm is initiated at time $t' > t$. D reaches a state in which no key is authorized to more than one node in finite time.*

We assume a stable logical bootstrap node, which always knows the IP addresses of some nodes in the DHT. Its only role is to initiate a round of the authorization algorithm every τ_{Token} seconds by contacting some DHT node, which we will denote by n_0. We can use standard distributed system replication techniques to make this bootstrapping a persistent service. We first describe the basic idea of the algorithm in a simple but deficient design and then discuss how we come up with a practical algorithm.

3.1 Simple but Impractical Algorithm

We describe a simple algorithm that authorizes nodes to keys in a DHT with a ring geometry (for example, Chord, Pastry) while satisfying key-consistency as an illustration of the basic ideas behind the authorization algorithm.

The authorization algorithm is initiated by node n_0 when it forwards a token to its successor in the ring. The token is forwarded by a node to its successor until it returns to n_0. n_0 begins with the entire keyspace $R = [0, 2^m - 1]$. n_0 removes its keyset $key(n_0) = [k_0, k_0^1)$ from R to obtain a range R'. n_0 is now authorized for $key(n_0)$. n_0 sends a token containing R' to its successor n_1. Now n_1 repeats this process by declaring itself authorized for $key(n_1) \cap R'$ and forwards $R'_1 = R' - key(n_1)$ to its successor. The removal of a key from the token once it is authorized to a node ensures that only one node can be authorized for a key. This algorithm is repeated to adapt to changes in the DHT. A node can remain authorized for a certain period after the latest authorization. This prevents it from being designated authorized forever. This consistency requirement leads to certain timing constraints, which we discuss in Section 3.3.

This algorithm is simple and it is applicable to all DHTs with a ring geometry, but it is not practical. It can suffer from low availability, does not scale well due to $O(N)$ time complexity, and is not robust in the face of packet drops. In the next section, we will present an algorithm that addresses these issues.

3.2 Basic Algorithm

We present the scalable algorithm in several stages. We use Chord as the underlying DHT. For simplicity, let us assume that a Chord node owns all the keys that lie between itself and its successor. Node n_0 (whose identifier is k_0) starts out with the entire keyspace $R = [0, 2^m - 1]$. Let n_0^1 (whose identifier is k_0^1) be the successor of n_0 and n_0^2 (whose identifier is k_0^2) be n_0's farthest finger. n_0 removes its keyset $key(n_0) = [k_0, k_0^1)$ from R to obtain a range R'. n_0 is authorized for $key(n_0)$. It then splits R' into two ranges $R_1 = [k_0^1, k_0^2)$ and $R_2 = [k_0^2, k_0)$. It passes R_1 to n_0^1 and R_2 to n_0^2. Now n_0^1 repeats this process by declaring itself authorized for $key(n_1) \cap R_1$ and splitting $R_1' = R_1 - key(n_1)$ amongst its successor and its farthest finger that lies in R_1'. n_0^2 does the same procedure. This *range-splitting* process is repeated till we reach a node that does not contain any further nodes in the region passed to it.

The range-splitting creates an *authorization tree* rooted at n_0. The information on the range handed down the tree is passed from node to node by an `authorize` token. By ensuring that no descendant in the authorization tree can claim a key authorized to an ancestor and by partitioning the keyspace at each level of the tree, we ensure key-consistency. In fact, we can use all the fingers available at a node to partition the keyspace.

We perform the authorize phase at regular intervals to adapt to the evolution of the DHT. The problem that arises is that a node that gets an `authorize` token may get partitioned from the DHT so that it does not receive any further tokens. This gives rise to a scenario in which there are stale `authorize` tokens in the DHT leading to possible inconsistencies. To solve this issue, we use a two-phase approach. In the first phase, the *collect* phase, every node receives a `collect` token and a range that it splits and forwards as described earlier. However, nodes do not declare themselves authorized in this stage. Instead, when a node sends out a `collect` token, it expects an ack from each of its children in the authorization tree.[3] When it gets acks from all its children or it has waited long enough, it prunes away those children that did not ack and then sends an ack to its parent in the tree. When node n_0 gets acks from all its children or has waited long enough, it begins the next phase, the authorize phase, by sending out the `authorize` token. However, nodes now forward this token only along the pruned tree.

```
token {
    src; // The node that initiates the token
    sender; // The node that sent the token
    sequence; // The sequence number of the token
    range; // The region of the keyspace allocated
        to the current node
    nextToken; // The time when the next token will be sent
    rt; // The time to wait for the authorize token
        corresponding to the collect token
    }
```

Figure 3: The format of a token

The period between the issue of consecutive `collect` tokens by n_0 constitutes a *round*. Each token carries a sequence number that is incremented for each round. The sequence number is used by nodes to discard old tokens. The token carries the time interval before the next token is sent out and the time to wait for the `authorize` token that follows a `collect` token. These values are used in setting the timers at the nodes (see Section 3.3). The token also carries the range that is delegated to the current node. The format of a token is shown in Figure 3.

Each node starts off in a NON-AUTH state. When a node gets a `collect` token, it goes into a WAIT state. When n_0 times out on the `collect` token or gets acks from all its children, it forwards an `authorize` token with the same sequence number as the `collect` token. A node in the WAIT state accepts the token only if it has the same sequence number as the latest `collect` token seen. Further, the `authorize` must have arrived within τ_s of the

[3]In the case where a Chord node owns keys between its predecessor and itself, the acks carry this additional information. Refer Appendix 6.

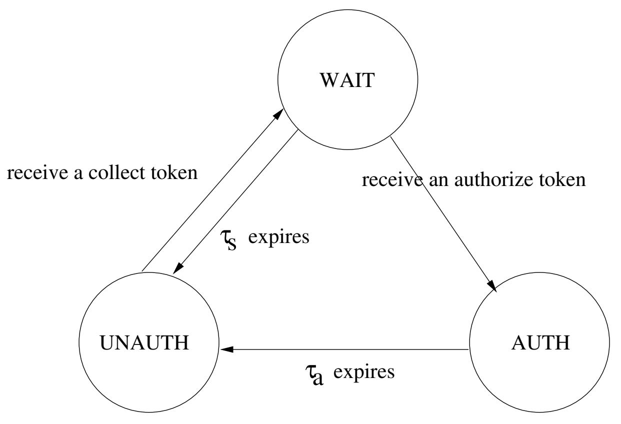

Figure 4: State machine of the authorization algorithm.

collect (the reason for this will become clear in Section 3.3). The node then moves into an AUTH state with respect to the keys in its keyset that are present in the authorize token and stays at the state for τ_a seconds. The state machine that represents the authorization algorithm is shown in Figure 3.2.

The modified authorization algorithm algorithm has a time complexity of $O(\log N)$ and an average per-node message complexity of $O(1)$. It is more resilient to message losses since the loss of a message does not always result in all nodes moving into the NON-AUTH state.

3.3 Timing

We make use of various timers to ensure that the protocol can provide guarantees about key-consistency. For simplicity, we only discuss setting these timers to constant values. The timers that control the authorization algorithm are:

- τ_{Token} - This timer controls the duration of a round. We could either set τ_{Token} to a large constant value. The value of τ_{Token} in the current round is referred to as T.

- τ_s - This is the time within which a node must receive the collect and authorize tokens in a round. By bounding the separation between the collect and the authorize tokens and adjusting the time spent in the AUTH state, we can avoid inconsistencies due to delayed authorize tokens and network partitions. In our application, we set $\tau_s = R$, where R is a time specified by node n_0 and communicated in each collect token.

- τ_a - This is the time for which a node can remain in the AUTH state. So we set $\tau_a = \tau_{Token} - 2R = T - 2R$ pessimistically so that even the last node to get an authorize token moves out of the AUTH state before the earliest node to get an authorize token in the next round. This again ensures that inconsistencies do not arise due to nodes missing the collect-authorize tokens. We set T and R so that the authorization algorithm responds quickly to changes in the DHT while nodes remain in the authorized state for as long as possible. For a given value of T, small and large values of R will cause nodes to switch to the NON-AUTH state though the correctness of the algorithm is unaffected. A reasonable value for R is the time taken by a token to return to n_0.

The authorization algorithm ensures that the DHT attains and, subsequently remains, in a key-consistent state. However, to be useful, the nodes must be authorized for keys that are conflict-free. If we consider the authorization algorithm in a perfectly consistent DHT with tokens having a deterministic time of transmission, each node stays in the AUTH state for a time $T - 2R$ and switches back to the NON-AUTH state till it is authorized in the next round. Each node is in the AUTH state for a fraction $\frac{T-2R}{T}$ of the time. A client intending to do a an atomic update would simply have to retry a lookup made during this time.

To tackle the problem of availability, we modify the algorithm so that when a node becomes authorized for a key for the first time, it waits in a PROVISIONAL state and then enters the AUTH state. However if it already was in the AUTH state for a key and the current round of authorization algorithm declares that it is the authorized root, then it simply remains in that state. The time for which a node remains in the AUTH state depends on whether the node is newly authorized for that key. τ_a^{old} is the time for which a node that retains its authority on a key remains authorized. τ_a^{new} is the time for which a node that is newly authorized remains authorized. τ_p is the time that such a node waits in the PROVISIONAL state.

Timer	Function	Value
τ_{Token}	The time period of the token	T-constant or decided by n_0
τ_s	Time spent in waiting for a test token after getting a set token	R-time between the two phases
τ_p	Time spent in the provisional state	$< T$
τ_a	Time spent in the AUTH state	$T - 2R + \tau_p$ for a previously authorized key
		$T - 2R$ for a newly authorized key

Table 1: A summary of the values used for the different timers

We set $\tau_p + \tau_a^{new} = \tau_a^{old}$. By choosing τ_p appropriately, we can ensure that a node does not oscillate between AUTH and NON-AUTH states during normal operations.

For example, we can set $\tau_p = W = R + S$. So any node that becomes newly authorized for a key will become truly authorized only after at least $R + S$ seconds. A node that was previously authorized on the same key and has become disconnected loses its authorization $R + S$ seconds into the next round ensuring that key-consistency holds. If there is no new claim on the key, the node that was formerly authorized will remain so if it receives the `authorize` token $R + S$ seconds into the next round. There is a trade-off between a value of τ_p which ensures that nodes do not oscillate between states during normal behavior, and one that makes nodes become authorized quicker under churn. We summarize the timers used and their values in Table 1.

3.4 Proofs of Correctness

In this section, we prove the following two theorems about the authorization algorithm.

Theorem 1 *In the DHT D, if no more than a single node is authorized for a key at time t, the key will be authorized to at most one node for all $t' > t$.*

Theorem 2 *Consider a DHT D in an arbitrary state at time t, i.e., more than one node may be authorized for a key. An authorization algorithm is initiated at time $t' > t$. D reaches a state in which no key is authorized to more than one node in finite time.*

Let K be the set of keys, N the set of nodes. We assume that clocks at the different nodes move at the same speed although they may not be synchronized.

Definition 1 *$K_t(x)$ is the set of keys owned authoritatively by node x at time t.*

The authorization algorithm CC comprises a collect and a authorize message. Denote the authorization algorithm with sequence number l by $CC(l)$. $collect(l)$ and $authorize(l)$ denote the collect and authorize messages with sequence numbers l. $collect(l).nodes$ is the sequence of nodes visited by the collect message. $authorize(l).nodes$ is the sequence of nodes visited by the authorize message. $CC(l).K$ is the set of keys which have unique authoritative roots. We denote by T_l the time elapsed between the issue of collect tokens $collect(l)$ and $authorize(l+1)$. R_l is the the time taken by the token to go around the ring used in round l. Both these values are communicated by n_0 to the other nodes through the token. The value of the timer τ_s used in round l is denoted by S_l. The time spent in the provisional state in round l is W_l, $W_l \leq T_l$.

Definition 2 *We refer to all the events that occur between the issue of two consecutive set tokens by n_0 as belonging to a round. So all events that occur since $collect(0)$ has been sent out and before $collect(1)$ has been sent out belong to round* 0.

For $k \in K_t(x)$ to hold, the precondition P_0^x must hold $P_0^x(k)$: The latest round of authorization algorithm in which node x participated must authorize $k \in K_{t'}(x), t' \leq t$. Let $Q_1^x(l)$: x has successfully completed a authorization algorithm $CC(l)$ at $t' \in [t - \tau_a, t]$. $P_0^x(k)$ is true if $P_1^x(k,l)$ is true where $P_1^x(k,l) : Q_1^x(l)$ and $\forall l' > l \neg Q_1^x(l')$ and $k \in CC(l).K$. For successfully completing the authorization algorithm, x must see a collect and authorize token:

1. Both of which have the same sequence number l'
2. Arrive at x within a time window $\tau_s = S$
3. that have traversed the same sequence of nodes.

Let $Q_2^x(l)$: x has seen a collect token $collect(l)$ at $t'' \in [t' - \tau_s, t']$ and a authorize token $authorize(l)$ at $t' \in [t - \tau_a, t]$ and $collect(l).nodes = authorize(l).nodes$. Here $collect(l).nodes$ and $authorize(l).nodes$ refers to the collect of nodes visited by $collect(l)$ and $authorize(l)$ respectively. We can ensure $P_1^x(k,l)$ is true by satisfying $P_2^x(k,l)$: $Q_2^x(l)$ and $\forall l' > l \neg Q_2^x(l')$ and $k \in CC(l).K$.

We can see from condition $P_2^x(k,l)$ how the algorithms for $set(token)$ and $test(token)$ look like.

Lemma 1 *Let $K_t(x,l)$ be the set of keys authorized to node x after participating in $CC(l)$. $K_t(x,l) \cap K_t(y,l) = \phi$.*

Proof: The collect token $collect(l)$ is sent out by node n_0 at time t and it returns to n_0 at some later time $t + \Delta_1$. First, we prove that no node other than those in $collect(l).nodes$ responds to the corresponding $authorize(l)$. This is based on two properties of the consistency check: the consistency check uses an increasing sequence number for each round, and every node that receives a token forwards it to only its successor and no other nodes. Consider a node $w \notin collect(l).nodes$ that responds to $authorize(l)$. Since n_0 uses increasing sequence numbers and the *collect* and *authorize* can only be τ_s apart, the last *collect* token received by w must be $collect(l)$. Thus $w \in collect(l)$. If there exists nodes $K_t(x,l) \cap K_t(y,l) \neq \phi$ and since a node forwards a token to only its successor, this implies that atleast one of the nodes did not receive $collect(l)$ which contradicts our earlier proposition. ■

Proof*(of Theorem 1)*: At time $t = 0$, the designated node n_0 sends out a collect token with sequence number l. We assume that at some time $0 \leq \Delta < T$, the DHT D satisfies proposition $P(t)$: $\neg\exists x,y \in N(t), K_t(x) \cap K_t(y) \neq \phi$. We prove that this proposition is satisfied for $\Delta < t \leq T_l$.

Let $\tau_a = \tau_a^{new} + \tau_p = \tau_a^{old}$. Let there be nodes x and y such that $K_t(x) \cap K_t(y) \neq \phi$. Thus, for some key k $P_2^x(k,l_x)$ and $P_2^y(k,l_y)$ must be true. From Lemma 1, $l_x \neq l_y$.

Consider $W_l \leq t \leq T_l$. Now consider the interval $[t - \tau_s - \tau_a, t]$. Node x must have seen a collect and a authorize token in this interval. This means that n_0 must have sent out a authorize token in this interval. If $\tau_a + \tau_s < W_l + T_{l-1} - R_{l-1}$ then $t - \tau_a - \tau_s > -T_{l-1} + R_{l-1}$. The only token sent out by n_0 during this interval is $authorize(l)$. Thus x and y receive tokens with the same sequence number l contradicting our assumption that $l_x = l_y$. Thus, we have $\tau_a < T_{l-1} + W_l - R_{l-1} - \tau_s$.

Consider $\Delta < t < W_l$. We now see that $t - \tau_a - \tau_s > t - T_{l-1} + R_{l-1} - W_l$. If $W_l < T_l$, we have $t - \tau_a - \tau_s > t - T_{l-2} + R_{l-2} - W_{l-1} > -T_{l-2} - T_{l-1} + R_{l-2}$. Thus, $P_2^x(k,l-2)$ cannot be true. Thus, the only authorize tokens that could possibly be sent out by n_0 during the interval $[t - \tau_s - \tau_a, t]$ are $authorize(l-1)$ and $authorize(l)$. However, no node could have been newly authorized in round l because $t < W_l$. If say $P_2^x(k,l)$ is true, then $P_2^x(k,l-1)$ must also be true. Thus, $P_2^x(k,l-1)$ and $P_2^y(k,l-1)$ must both be true contradicting the assumption that $l_x \neq l_y$. From this, we have $W_l < T_l$. ■

Proof*(of Theorem 2)*: Let $t' = t + T$. A node that authoritatively owns a key at t' must have participated in a consistency check between t and t' since $T > \tau_a$. If we have two nodes x and y that authoritatively own key k at t', node n_0 must have sent out a authorize token between t and t'. This contradicts Lemma 1. ■

The value of τ_p can be collect to ensure that the nodes do not lose authorization frequently under normal conditions. For example, in round l, we can collect $\tau_p = W_{l+1} = R_{l+1} + S_{l+1}$. So any node that acquires authorization for a key in round $l+1$ will become authorized only after atleast $R_{l+1} + S_{l+1}$ seconds. A node that was previously authorized on the same key and has become disconnected loses its authorization $R_{l+1} + S_{l+1}$ seconds into round $l+1$ ensuring that K-consistency holds. If there is no new claim on the key and the node that was formerly authorized does not get disconnected, it will remain authorized if it receives authorize token $authorize(l)$ $R_{l+1} + S_{l+1}$ seconds into round $l+1$.

The question that now arises is: how do we predict the values $R_{l+1} + S_{l+1}$ to be used for τ_p. The correctness of the algorithm is independent of the value chosen for τ_p as long as the value of $\tau_p < T_{l+1}$. This can be enforced by node n_0 which decides to use the decide the value of T_{l+1} based on the current value of τ_p. In fact, the value of τ_p can be kept approximately constant provided it satisfies the inequality $\tau_p < T_{l+1}$. Or if a more accurate estimate is needed, we can use an estimate of the time taken for the token to go around the consistency in the next round. There is a trade-off between a high value of τ_p which ensures that nodes do not oscillate between AUTH and NON-AUTH states during normal behavior, and a low value that makes nodes become autho-

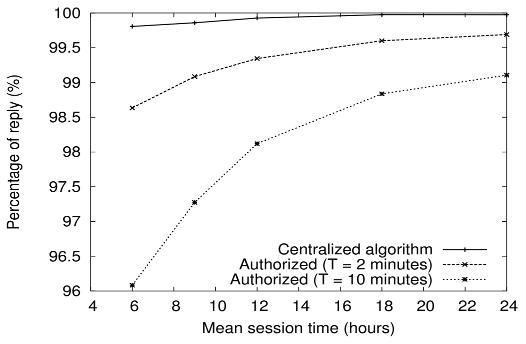

Figure 5: Effect of mean session time on availability.

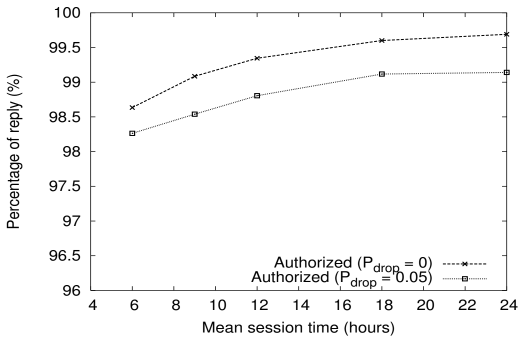

Figure 6: Effect of message loss on availability.

rized quicker under churn. We summarize the timers used and their values in Table 1.

4 Experiments

In Section 3, we have shown how the authorization algorithm provides key-consistency guarantees while retaining scalability and a low overhead of operation. To be useful, the authorization algorithm must also provide high availability *i.e.*, it must ensure that a high fraction of the keyspace is authorized and that the DHT routing algorithms reach the authorized root as often as possible. In this section, we measure the availability of the authorization algorithm.

We have implemented Chord and the authorization algorithm. The first set of experiements were done on a PlanetLab deployment to see how our algorithm performs in the wild. We then performed measurements on a simulated network to evaluate the behavior of the algorithm under churn and in the face of packet drops.

We performed measurements on the availability of our key-consistency mechanism in a deployment of approximately 300 PlanetLab nodes. A token is issued every four minutes. Each node generates a lookup for a random key. The interarrival time of these lookups is exponentially distributed with a mean of one minute. The lookups reached the authorized roots in 97.6% of the queries which is higher than the percentage of the uncontested keyspace presented in Figure 1. In almost all of these failure cases, there was an authorized root for the key, but the DHT routing led to a root that wasn't authorized. We can improve availability further by improving the Chord routing algorithm, but this is not the focus of our study. Any improvement in routing will increase the availability.

The simulation testbed comprised a network of 500 nodes. We induce churn in the network. Each node stays in the network for an exponentially-distributed session time with mean T_S equal to 6, 9, 12, 18, and 24 hours respectively. Each time a node leaves the network, we add another node to maintain the total number of nodes to be 500. Each node generates lookups for a random key. The interarrival time of the lookups is exponentially distributed with a mean of one minute.

Figure 5 shows the availability (percentage of replies from authorized roots) of the authorization algorithm with varying mean session time. The two graphs correspond to token period $T = 2$ minutes and $T = 10$ minutes respectively. With a 2-minute token period, the algorithm can guarantee 98.5% availability with a 6-hour mean session time. As the mean session time decreases, the availability decreases especially when the token period becomes higher. When a node fails after getting a collection token, the subtree rooted at the node will not be authorized until the next token is issued. By reducing the token period, we can limit the effect of churn on availability. As a reference, we show the percentage of replies that reach an authorized root as determined by a central authorization algorithm. This is the best possible we can achieve, since it elides transient network outages, non-transitivity, and message loss. Our authorization algorithm achieves availability close to that of the central algorithm.

Figure 6 shows that the authorization algorithm is resilient against lost messages in the simulated network, since lost messages are retransmitted. When 5% of the messages are dropped in the network, and the token period is two minutes, the availability decreases only by 0.5%.

5 Related Work

Emulating atomic shared memory in a distributed setting is a widely studied problem. The instantiation to DHTs was discussed in [6]. The solution assumes that nodes execute a leave procedure before leaving and that churn is the only cause of multiple roots. Rosebud [10] is a Byzantine fault tolerant distributed storage system that can change replica configuration under dynamic membership. Etna [7] supports atomic updates by using Paxos over replica sets to handle changes in replica set configurations. The problem of multiple roots remains and our approach can be used as a basic substrate on which to layer such approaches. The problem of multiple roots has been considered in [1] and [4]. The former proposes a solution to the problem in the face of churn using a consistent join protocol. The latter fixes the problem of multiple roots due to non-transitive links. Both these approaches, however, are probabilistic in nature.

6 Conclusion

DHTs present a challenge that is ever present in distributed systems; strong data consistency guarantees are hard to achieve at scale. Our paper ducks this fundamental challenge by considering key consistency rather than data consistency. However, we argue that key consistency enables one to build DHT primitives with atomic semantics, something that is difficult in traditional DHTs. Moreover, we achieve key consistency in a way that does not interfere with the underlying DHT routing algorithm; one is free to optimize DHT routing without endangering the key consistency guarantees. Thus, our approach does not impose any performance penalties on applications which do not use the key consistency properties; the application merely ignores the *auth* bit.

While atomic primitives do not ensure data consistency, we hope to show in future work how clients can use algorithms such as those in [3, 5] to achieve data consistency with client-side algorithms.

References

[1] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *DSN*, 2004.

[2] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A case study in building layered DHT applications. In *SIGCOMM*, 2005.

[3] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *PODC*, 2002.

[4] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *WORLDS*, 2005.

[5] E. Gafni and L. Lamport. Disk paxos. In *DISC*, 2000.

[6] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *IPTPS*, 2002.

[7] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. In *Techinical Report, MIT*, 2005.

[8] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, 2003.

[9] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, , and H. Yu. OpenDHT: A public DHT service and its uses. In *SIGCOMM*, 2005.

[10] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. In *Technical Report, MIT*, 2003.

A Pseudocode for authorization algorithm

```
INITIATE()
1 $token.src \leftarrow id$
2 $token.sequence \leftarrow sequence++$
3 $token.level \leftarrow 0$
4 $token.range \leftarrow [0, 2^m - 1]$
5 $collect(token)$
6 $set\_timer(NEXT\_ROUND, \tau_{Token})$

COLLECT($token$)
 1 if $token.sequence > sequence$
 2   then $children \leftarrow \phi$
 3        $sequence \leftarrow token.sequence$
 4        $level \leftarrow token.level$
 5        $parent \leftarrow token.sender$
 6        $state \leftarrow WAIT$
 7        $token.sender \leftarrow id$
 8        $token.level++$
 9        $token.range \leftarrow token.range - my\_region$
10        $F \leftarrow$ ( fingers with increasing ids $\in token.range$)
11        if $F == \phi$
12          then $parent.ack(id, sequence, my\_region)$
13               $set\_timer(WAIT\_FOR\_DELEGATE, \tau_s)$
14          else for $f \in F$
15                 do
16                    $new\_token \leftarrow token$
17                    $next(f) \leftarrow$ closest succeeding finger of f
```

```
                        $new\_token.range \leftarrow$
                            $[f, next(f) - 1] \cap token.range$
                        $f.collect(new\_token)$
                $set\_timer(WAIT\_FOR\_ACKS, \tau_w)$

```

```
ACK($id', sequence', range$)
if $sequence' == sequence$
  then $children \leftarrow children \cup id$
       $range(id) \leftarrow range'$

if $children == F$
  then if $level == 0$
          then $token.src \leftarrow id$
               $token.range \leftarrow [0, 2^m - 1]$
               $authorize(token)$
          else $parent.ack(id, sequence, my\_region)$
               $set\_timer(WAIT\_FOR\_DELEGATE, \tau_s)$
```

```
AUTHORIZE($token$)
if $token.sequence == sequence \&\& state == WAIT$
  then $provisional \leftarrow my\_region \cap token.range - authorized$
       $set\_timer(AUTHORIZE, \tau_w)$
       $authorized \leftarrow my\_region \cap token.range \cap authorized$
       $set\_timer(DE-AUTHORIZE, \tau_a)$
       $state \leftarrow AUTHORIZED$
       $token.range \leftarrow token.range - my\_region$
       for $f \in children$ in increasing order of ids
         do $new\_token \leftarrow token$
            $new\_token.range \leftarrow$
                $[f, next(f)] \cup range(f) - range(next(f))$
            $new\_token.range \leftarrow$
                $new\_token.range \cap token.range$
            $authorize(new\_token)$
```

```
HANDLE_TIMER($event$)
switch
case $event = NEXT\_TOKEN$ :
     $initiate()$
case $event = WAIT\_FOR\_ACKS$ :
     $parent.ack(id, sequence, my\_region)$
     $set\_timer(WAIT\_FOR\_DELEGATE, \tau_s)$
case $event = WAIT\_FOR\_DELEGATE$ :
     $state \leftarrow UNAUTHORIZED$
case $event = AUTHORIZE$ :
     $authorized \leftarrow authorized \cup provisional$
     $provisional \leftarrow \phi$
     $set\_timer(DE-AUTHORIZE, \tau_a - \tau_w)$
case $event = DE-AUTHORIZE$ :
     $state \leftarrow UNAUTHORIZED$
     $authorized \leftarrow \phi$
```