Mojito under Churn

Jay Liu
Department of Computer Sciences, The University of Texas at Austin
jayliu@cs.utexas.edu

May 2, 2009

Abstract

In a dynamic hash table (DHT), peers in a network cooperate to maintain a distributed key-value storage. Each peer in a DHT is responsible for storing a subset of keys and their values. Especially in a peer-to-peer system, the peers may not be stable. New peers join and existing peers either fail or leave the system. This process, called churn, requires the DHT to be able to reorganize itself. This paper looks at the performance of a specific DHT, called *Mojito*, as peers join and leave.

1 Introduction

Dynamic hash table (DHT) is a distributed version of the hash table data structure [6]. In a distributed system environment, multiple peers cooperate to store data. Each peer is responsible for maintaining entries for a subset of keys. In such a system, peers may not be persistent. Existing peers may fail or go offline at any time, and new peers may join the network. This continuous process of arrival and departure of network nodes is called "churn" [7].

Churn causes instability in a DHT. A peer who was previously being relied upon to provide storage for a specific key may no longer be available. This responsibility needs to be handed off to another peer, either proactively or reactively. Since the peer may crash unexpectedly, proactive handoff cannot be guaranteed, so the network must possess a reactive mechanism. In a reactive system, when other peers need to access this key, they need to discover the location of the new responsible peer on their own by exploring the network.

Churn as applicable to the DHT had been previously studied. A survey in [7] gives an overview of how popular recursive routing DHT implementations handle environments with high churn. Routing methodologies will be explored in Section 2.1.

In this paper, we will describe *Mojito* [4], which is an implementation of the Kademlia DHT. It is used by the popular LimeWire Gnutella peer-to-peer (P2P) software. Kademlia was introduced by [5] as a theoretical system, and Mojito is one of a number of implementations of it. Although being widely deployed with LimeWire, little experimental study had been performed on Mojito to evaluate its performance.

Since Mojito is deployed on a P2P system, where peers can join or leave without regularity, churn rate is an important metric that directly affects the stability of the DHT. P2P systems typically experience high churn rate [8]. A system that does not adapt well to high churn rate means an application needs to more strictly control who gets permission to participate in the system. Such restriction means that not all peers will immediately benefit from the DHT. We will therefore perform experimental evaluation of the impact of churn rate on Mojito.

We will simulate peers joining and leaving, and observe the correctness and performance of the operations of the DHT. Multiple standalone Mo-

jito instances will be created. New instances will be randomly added, and existing instances will be randomly removed to simulate churn. After each such action, we attempt to retrieve a key-value pair that was previously in the DHT. This may succeed quickly, experience a delay, or simply fail. The experiment will observe any changes in the success rate of lookup requests as churn rate varies.

This paper is organized as follows. It first provides a description of general DHT characteristics. It proceeds to describe in more detail Kademlia and Mojito. It also discusses existing understanding of churn and its application to P2P system. Experiment setup will be described, along with the results. It will conclude with an overview of related and future works.

2 DHT

2.1 Mojito

Each value in a DHT is associated with a key. The key is used to find the peer that is responsible for storing the value. A peer typically can hold values for multiple keys because the number of possible keys is usually much larger than the number of peers in a system. It is possible that multiple peers share the responsibility for a key. This provides redundancy. However, since the peers may not be equally reachable in a network, the values for the key at different locations may go out of sync. In a unstable system, it may also be possible that none of the active peers claim responsibility for a key, potentially leading to data loss.

Primary operations in a DHT include `put(key, value)`, `get(key)` and `remove(key)`. All operations are similar in that given a key, they all need to identify a peer that owns that key. There are different ways to divide key ownership among peers. CAN [6], for example, constructs a global coordinate space, and each peer is assigned a portion of that space. The keys are coordinates in the global space. The peer whose subspace is the one in which the key's coordinate falls is then responsible for that key.

In Kademlia, both peer IDs and object keys share a linear ID space. The peer whose ID is the closest to a given key is then responsible for that key.

It can be observed that in CAN, if a peer goes offline, a subspace becomes vacant, and requires neighbor peers to be aware of this hole and work together to fill it. In Kademlia, the peers do not need to move, since as long as there is at least one peer, there is always a peer whose ID is the closest to any given key.

Each Kademlia peer uses a 160-bit SHA-1 hash value as its identifier. The IDs are randomly generated by each peer, using a combination of pure randomness and hashing peer attribute values such as IP address. Gaps or overlaps, when they do exist, do not hinder the system. The object keys are hashed to share the same 160-bit ID space as the peer IDs. Since object keys share the same identifier space as the peer IDs, the distance between two object keys, or between a key and a peer, can also be calculated.

To find the closest peer, a distance function is needed. Kademlia uses XOR operation to measure distance between IDs [5]. The distance d between two IDs x and y is defined as:

$$d(x, y) = x \oplus y$$

The smaller the distance d, the closer two IDs are. XOR can be used as a valid distance function due to these properties:

$$d(x, x) = 0$$

$$d(x, y) = d(y, x)$$

$$x \neq y \leftrightarrow d(x, y) > 0$$

$$d(x, y) + d(y, z) \geq d(x, z)$$

where the last property is the triangle inequality.

In XOR distance, the more most-significant bits two IDs share, the closer they are to each other. To take advantage of this, routing tables in Kademlia are stored in a unbalanced trie

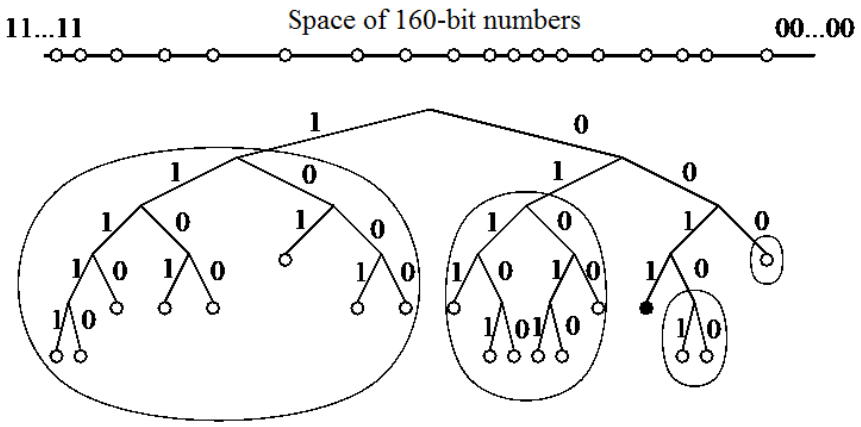

Figure 1: Kademlia routing table trie for node with prefix "0011...", which maintains contact with at least one node in each circled subtrie [5].

structure, as shown in Figure 1. Peers that have short XOR distance are close in the trie as well.

Another unique feature of Kademlia is that it uses an iterative routing algorithm, which sets it apart from the recursive routing scheme employed by other DHTs. The two approaches are illustrated in Figure 2. In general, routing is necessary in DHT because each peer only knows a limited number of neighbor peers. The requesting peer may not have contact information about the destination peer. Instead, it would pick some of its known neighbors that are the closest in distance to the destination. The idea is that some of these neighbor nodes may be able to reach the destination, or at least get even closer to the destination.

In an iterative routing algorithm, the original requesting peer p_i is always the peer making requests. Let p_j be one of its neighbors. For a request from p_i, p_j will return a result set along with a list of its neighbor peers, e.g. p_k. If p_j is not the destination peer, the result set is empty. Then, p_i will continue its request by directly contacting p_k itself. In a recursive routing scheme, p_j would contact p_k on behalf of p_i instead.

Theoretical analysis performed in [11] between DHTs that route recursively and those that route iteratively showed that iterative routing is much more resilient to churn than recursive routing.

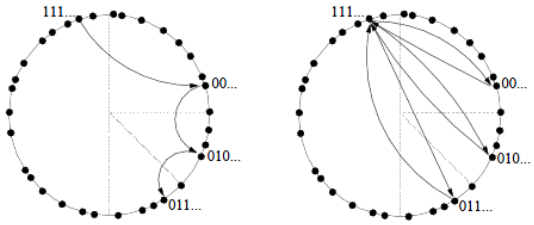

Figure 2: Recursive (left) and iterative (right) routing schemes [7]. The requesting peer is "111...". The destination is "011..." but it is not directly reachable from requester. The requesting node uses one of its known neighbors "00..." to approach the destination node.

One of the benefits of iterative routing is that the time-out threshold at the requester can be set to exactly 2 hops, i.e. the round-trip time (RTT). A hop is defined as one direct communication from one peer to another. In contrast, the time-out threshold for recursive routing at the requester must allow for the maximum number of hops possible to find the value, which is at least 2 hops. In a system that is prone to time-outs, such as during times of high degree of churn, iterative routing allows quicker time-outs. This frees up the requesting peer to perform additional tasks.

There are two useful parameters that affect the

performance of Kademlia: α and K. α defines the number of neighbors a peer contacts concurrently for a lookup request. Experimental results indicate a value of 3 is sufficiently optimal [9]. Parameter K defines the size of the routing table "bucket". There are 160 such buckets. Each bucket i stores contact information about neighbor peers that are between 2^i and 2^{i+1} units of distance away from the current peer. K is set to 20 by default. A larger K value means more contacts are stored, providing backup peers in case of failures. Unfortunately, it would also increase overhead. Due to churn, information about peers change quickly. Having more peers requires more maintenance work to keep contact information up to date.

2.2 Churn

In a dynamic system like P2P, the membership varies over time. This is characterized as churn. New peers may join at any time. These peers provide resources that can take advantage of the DHT. They provide new values to share. They can also contribute to DHT routing tasks. On the other hand, peers may leave the DHT at any time. This can be caused by deliberate exit, or a system failure at the peer.

Both events can be difficult to predict, and each such event causes changes in the DHT. When a peer p_i leaves the system, many other peers may still have it has a contact. This leaves some routing tables with an incorrect entry. Given a key, a remaining node q may believe that p_i is the nearest node to the key, and will then attempt to contact p_i. Since p_i is down, this will lead to a long wait, resulting in a time-out. q will then have to traverse the network to find the correct node, which will take additional time. This degrades the lookup performance, and prevents q from servicing additional requests.

When a peer p_j joins the system, it creates a random identifier for itself. To be part of the system, it needs to locate and communicate with other peers to participate in the DHT. It needs to let others know its identifier so that others will route the appropriate keys to it. This is called the *bootstrap* process, and may take some time. A peer that joins experiences a delay before it is fully active, and other peers will need to help it in the bootstrap process. p_j may now be closer to some keys than the existing peers. It will take time for other peers to notice this, and replace their existing entry in the routing table with p_j.

Both churn events result in performance degradation. In conjunction with unreliable network connections between peers, they present great challenges to DHT design.

Churn C, normalized over a period T, is formally defined as [1]:

$$C = \frac{1}{T} \sum_{e_i \in E} \frac{|U_{i-1} \Delta U_i|}{\max\{|U_{i-1}|, |U_i|\}}$$

where events $E = \{e_i | e_i \in \{\text{join}, \text{leave}\}\}$. U_i is the set of active nodes after event e_i, and Δ is the symmetric set difference operator. Churn is proportional to the fraction of peers that have changed state, and inversely proportional to time. It is highly desirable to have peers that do not change their state, from being active to inactive and vice versa. If they must change state, they should remain in that state for as long as possible.

To avoid churn problems, one solution is to restrict membership. This is adopted by Mojito. A node only becomes an active contributing peer in the DHT after exhibiting consistently good uptime. A trade-off is made here. A higher uptime threshold makes the DHT more stable. Peers who have stayed around a long time tend to stay longer than those who have not [10]. However, a lower value promotes greater participation by the nodes. More nodes actively participating in the system also distributes workload among the participants as each participant will be responsible for fewer keys.

To cope with churn, there are three types of peers in a Mojito system [4]:

1. *Active*: This peer is an active participant. It provides DHT lookups and storage services,

and can make full use of the capabilities of the DHT. It has been fully bootstrapped.

2. *Passive*: This is a special mode reserved for "ultrapeers" who are able to provide DHT, but do not due to resource constraint. They can still request lookups from active peers, but do not participate in DHT routing itself.

3. *Passive leaf*: This is a peer that either cannot or chooses not to provide DHT services. The client may be behind a firewall, or is too new or unstable to qualify for being active. They do not participate in DHT, but can get data from the DHT by connecting through an ultrapeer who is in passive mode. Obviously, the number of nodes in this mode is restricted since these peers only add load to the system without contributing any resources.

A node is required to have an uptime of at least 2 hours before it is allowed to become an active peer. In addition, this node should have exhibited similar, good uptime behavior during prior sessions as well.

3 Experiment Setup

3.1 Mojito Network

A total of N peers, each peer with its own instance of Mojito, are brought online. The first peer brought online is not initially bootstrapped, since there are no other peers yet online. Each subsequent peer coming online will randomly pick one of the already online peers as its bootstrapper. It should be noted that as peers come online, the earlier a peer comes online, the more bootstrapping responsibility it has. This requires the earlier peers to be more stable than newer ones. Lastly, the first peer is bootstrapped using any of the other $N - 1$ peers now online.

In a production system, a peer needs to know another existing peer to join the system. It is important for different nodes to bootstrap randomly. Picking the same peer for bootstrapping duties for all other peers will cause instability when the bootstrapper peer contains errors in its routing table, or fails. During the bootstrap phase, a new node should discover as much about the network as possible.

After all N peers have been bootstrapped, the DHT is now running. It is populated. A set Y of $3N$ keys were randomly generated. Each key is associated with a random value, with which a random peer performs the `put(key, value)` operation to store the pair into the DHT.

Separate threads are then created to perform two separate tasks in parallel:

- *Churner:* This task simulates churn activity in DHT. Peers are randomly taken offline and replaced by new peers, corresponding to a peer leaving and another peer joining. It also calls a task called *publisher*, which introduces new values for existing keys into the DHT. These two tasks are discussed in Sections 3.2 and 3.4, respectively.

- *Prober:* During initialization, key-value pairs were put into the DHT. This task periodically performs lookup for values given a key. The key is randomly selected from Y. This is discussed in Section 3.3.

3.2 Churner

Two random variables govern churn activity. λ_J is the rate at which new nodes join the network, and λ_L is the rate at which existing nodes leave the network. In general, $\lambda_J \geq \lambda_L$, otherwise the network will steadily lose nodes over time and die out. For the simplicity of having a stable network of constant size N, the experiments use $\lambda_J = \lambda_L \equiv \lambda_C$. Whenever a peer leaves the network, a new peer is immediately created in its place. This replacement is not instantaneous, however. Bootstrap time will cause a delay before a new peer becomes fully functional. During this time, the network is missing participants, operating only at size $N_C \leq N$. At high churn rate, $N_C \ll N$.

Churn activity is modeled after a Poisson distribution, i.e. random and bursty, at a rate of λ_C. The event interarrival time τ is exponentially distributed. For the experiment, the median node interarrival time τ_{median} is varied to achieve the desired churn activity rate. It has the relationship:

$$\lambda_C = \ln(2)/\tau_{\text{median}}$$

It is easy to express the median session time t_{median} of the nodes in the network:

$$t_{\text{median}} = N\tau_{\text{median}}$$

3.3 Prober and Consistency

The prober is able to pick a random active peer, and a random key from Y, and perform a lookup using the `get(key)` operation. The peer and the key are chosen randomly and independently from each other, so the peer may or may not have previously interacted with this chosen key. Mojito returns an empty set if no value is found, a single result value if there is only one value associated with the key, or an empty result set with a list of secondary keys in case multiple values are available for a key. The secondary keys were automatically generated to identify multiple values. A secondary key must be used together with the primary key to retrieve the actual value. For each secondary key, the prober performs an additional lookup to get the actual value found in the DHT system. These lookups using secondary keys are herein referred to as "secondary lookups" to differentiate them from the initial lookup, which will be called "primary lookup".

The consistency experiment is set up similarly to [7]. The success metric with which we judge churn resiliency is the number of consistent lookup results over total number of lookup results. The probe experiments run in rounds. One round of probing dispatches 10 separate primary lookups for the same primary key, but each with a different randomly chosen active peer. Mojito allows a key to be associated with multiple values. By creating and dispatching any necessary secondary lookups, all values are returned. Since all keys in Y has at least one value stored in the DHT, lookups will succeed with some value unless there is data loss or routing error due to instability caused by churn.

If a value is returned by at least 5 of the 10 lookups, then those lookups that returned this value are considered to be consistent, while others that did not are inconsistent. If a value is returned by fewer than 5 lookups, lookups that returned that value are inconsistent, while the lookups that did not return the value form the consistent set. When multiple values are returned by a lookup, each value is evaluated separately.

The rate at which we initiate a round of lookups also follows the Poisson distribution, at a rate of:

$$\lambda_P = 0.1N/\text{second}$$

The cumulative consistency percentage is recorded after convergence is observed. Convergence is reached more quickly at low churn level. For each level of churn, the test is run until one of the following criteria is reached:

1. After running for at least 10,000 rounds, the consistency rate does not vary by more than 0.01% in 10 consecutive rounds. Since each round has 10 primary lookups, there are at least 100,000 different primary lookups. Only low levels of churn satisfy this strict convergence criterion. This should take fewer than 30 minutes to complete, but may take more.

2. While criterion 1 is ideal, it is achievable only at low churn rate. At high churn rate, it becomes very difficult to meet. In high churn situations, consistency rates exhibit greater fluctuations, and lookups often time out. The constraint needs to be relaxed such that convergence is accepted if in the last hour of execution, the rate does not drop by more than 1%.

3. The system may collapse under churn, where the number of active peers remaining $N_C < 10 \ll N$. A system that is unstable will reach this point quickly. This can occur despite having $\lambda_J = \lambda_L$. After a node arrives, it needs additional time to bootstrap before becoming active. During bootstrap time, additional nodes may have left, creating an imbalance.

Lookup rounds are not run in parallel, but sequentially. At high churn rate, dispatching multiple lookups in parallel, some of which can be very slow, overwhelms the system. At low churn rate, lookups are quick (1 ms, from Section 4.3) as compared to the interarrival time, and therefore, concurrency is not necessary.

3.4 Publisher

A third task, *publisher*, performs `put(key, value)` operations using random values for keys in Y. It is responsible for populating the DHT system. It is used both at startup and by the churner.

On startup, all keys in Y are published in the system such that each of the N peers is given the responsibility to publish 3 of the $3N$ keys in Y. As nodes leave and are replaced by new nodes, each new node will republish the 3 keys held by the node it replaced, using new random values.

Therefore, each join event adds 3 additional values to the DHT, 1 per key. The number of join events increases as the system ages. The longer the system stays up, the more values get associated with each key. While Mojito purges older values periodically, at high churn rate, many values may still be associated with a key despite such maintenance sweep. This causes the number of secondary lookups for keys to increase since each value needs to be fetched individually. Also, since there are more values, an unstable system is more likely to experience lookup inconsistency.

The purpose of this setup is to simulate a system where there are Y popular items, and each new peer that joins provides new contributions to three of the items in Y.

3.5 Performance Measurement

Lookup consistency is an important metric used to evaluate the DHT under churn. Looking at the consistency result alone, however, does not give a complete picture of the system. It is also important to examine the performance of some basic functions of the DHT since they may explain or predict the consistency results.

Under investigation are the costs of bootstrapping to the network, adding data, and finding data in the DHT. We keep timestamps for the operations under investigation with millisecond accuracy, and observe any changes or trends.

4 Results

4.1 Bootstrap Time

When a node comes online, it goes through a bootstrap process. In a bootstrap process, a node pings an existing node in the network, and tries to learn about its contacts. It may use those contacts to learn about additional nodes. A new node is only operational after bootstrap process is complete. Therefore, the amount of time the bootstrap process takes limits the rate at which new nodes can be introduced to the system. Leaving the system is typically much quicker than joining; in fact, leave is immediate in a crash scenario. The experiment must allow for bootstrap time. Otherwise, the system will lose nodes faster than it creates new nodes, causing it to collapse.

Figure 3 shows the amount of time it takes to bootstrap given the number of nodes in a system. All nodes are on a single host. In a small network, bootstrap is very quick. However, with an increasingly large number of nodes, bootstrap takes much longer time to complete. The median can reach up to thirty seconds. Sometimes, bootstrap takes well over a minute.

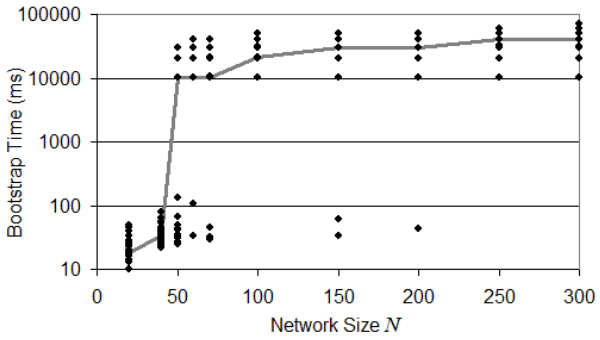

Figure 3: Bootstrap time for a new peer using peers on the same host; line connects the median values.

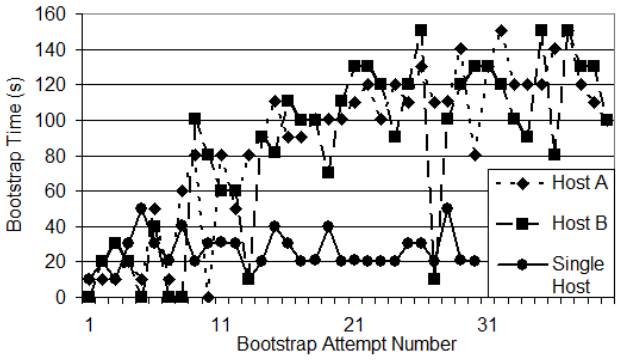

Figure 4: Bootstrap time over the network.

In a production system, bootstrap would actually take place over a network that is often unreliable. To observe the impact of network latency on bootstrap time, we used two hosts, each of which is running 50 instances of Mojito, and compared their bootstrap times during churn against a setup where a single host is running all 100 instances. Figure 4 shows that bootstrapping over the network eventually stabilized at 2 minutes, significantly higher than the 20 seconds needed by running all instances on a single host.

Therefore, by running all peers on the same host, the experiment setup is idealized. It is reasonable to expect any performance differences during churn to magnify in a production deployment.

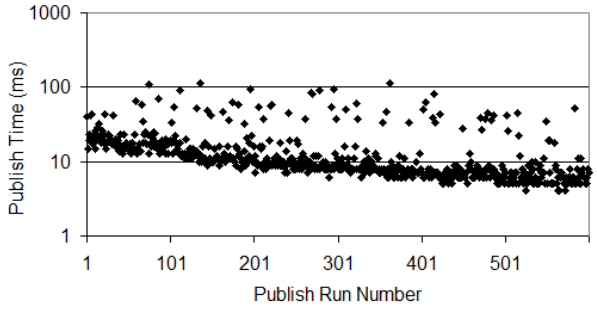

Figure 5: Publish time for 600 values before churn.

4.2 Publish Time

Publish time records the performance of the `put(key, value)` operation. A node is online after the bootstrap process completes. It will then publish three values into the system. The performance of storing key-value pairs in the DHT reflects the activity level of the system.

Running on a DHT of size $N = 200$ and $\tau = 10$ s, the first set of data, consisting of the 600 publications as shown in Figure 5, was collected during initialization, before churn activity has started. This is a period of system stability. At the beginning, publishing took over 10 ms. As more values were published, the system became more efficient. The overall median time was 9.5 ms. Some data points were significantly higher, causing the average to reach 14.8 ms.

The second set of data was collected during churn. This is shown in Figure 6. It can be observed that the publish time increased drastically, with majority of them taking at least 10 s.

While publishing values into the system is not necessarily a time sensitive operation to the user, these results clearly show that churn significantly degraded the performance of the DHT.

4.3 Lookup Time

Lookup time records the performance of the `get(key)` operation. Since large number of lookups are performed, primary lookup perfor-

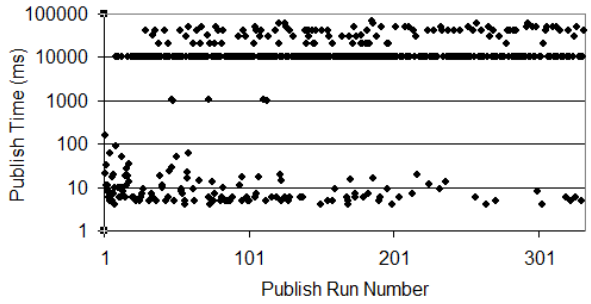

Figure 6: Publish time for 330 values during churn.

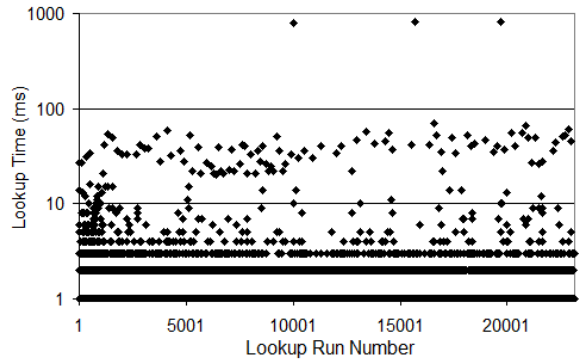

Figure 7: Lookup time for 23000 lookups.

mance was evaluated using $N = 200$ and $\tau = 10$ s.

In general, primary lookups were very quick, with a median of 1 ms, averaging 1.3 ms. It was only slightly impacted by churn activities. This is shown in Figure 7. One explanation is that data is well-replicated in Mojito. By default, Mojito stores the key-value pair not only at the destination peer, but also at 19 other peers that lie on the various routing paths to the destination. A lookup, therefore, does not necessarily need to find the exact destination node. It is sufficient to hit any one of the twenty peers. While such replication enhances lookup performance, it increases the risk of data inconsistency.

It is important to note, however, that many lookups performed worse than the median value, and some considerably worse. Lookups were set to time out after 1 second so a few timed out. In addition, these times represent a single primary

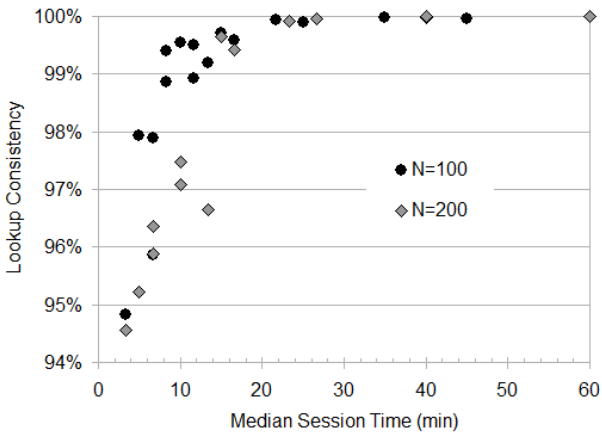

Figure 8: Lookup consistency result.

lookup. As mentioned earlier, a key can be associated with multiple values. This is especially important as the experiment ages, since each replacement node adds new values to 3 existing keys, but the values expire slowly by default. In these cases, the first lookup only returns a list of secondary keys. For each of the secondary key, a separate lookup needs to be performed. Therefore, the true cost of looking up all the values for a key is the shown value multiplied by the number of different values associated with the key, which is increasing as the system ages. Also, the more lookups need to be performed, the more likely it is to experience timeouts and inconsistencies.

4.4 Lookup Consistency

For lookup consistency, we ran the experiment multiple times, varying the rate of nodes leaving and joining the system, λ_C. Two networks, one of size 100 and another of 200, are used to show whether there exists differences between networks of different sizes.

In a stable system, without any churn activity and $N = 200$, equilibrium lookup success rate is 100%. This is the baseline. Any deviation from it is then solely due to churn. The lookup consistency results are presented in Figure 8.

There are three sections in lookup consistency.

When median session time is high, churn is low, lookups are very consistent, nearing 100%. The new node is able to start up completely, and is given enough time to fully stabilize in the system. Such stability is achieved for session times of at least 20 minutes. This setup allows the network to satisfy convergence criterion 1 from Section 3.3.

In the second section, between session times of 20 minutes and approximately 3 minutes, the system becomes increasingly unstable. Some could not satisfy convergence criterion 1, only criterion 2. There is an obvious drop in consistency rate when compared to the first section. The larger network experiences a greater level of degradation due to its higher bootstrap time, as discussed in Section 4.1. A higher bootstrap time means a peer requires more time to stabilize, so given the same session time, a smaller network performs better. In practice, LimeWire has many millions of peers over unreliable networks. This experiment shows that even in the most ideal case, a session time of less than 20 minutes is undesirable. On a Gnutella network, more than 20% sessions lasted fewer than 20 minutes [8]. For Mojito to function without consistency errors, this significant portion of the peers cannot be allowed to participate in DHT.

The worst case occurs when session times are shorter than 3 minutes. The experiment could not be performed correctly because the nodes are taken offline at a much quicker rate than new nodes are able to join the network. The system rapidly becomes unusable, with no nodes left after fewer than 20,000 lookups. No meaningful results were collected since the consistency rates were still changing rapidly when the system failed completely.

4.5 Bootstrap Lag

A metric closely associated with lookup consistency is the number of unbootstrapped peers, i.e. $N - N_C$. At any given time during churn, new replacement peers are being bootstrapped after the old peers were taken down. As shown in Section 4.1, bootstrap may take considerable amount of time. While peers are bootstrapping, the number of active peers is less than the total number of peers. A system that bootstraps quickly creates a more stable system.

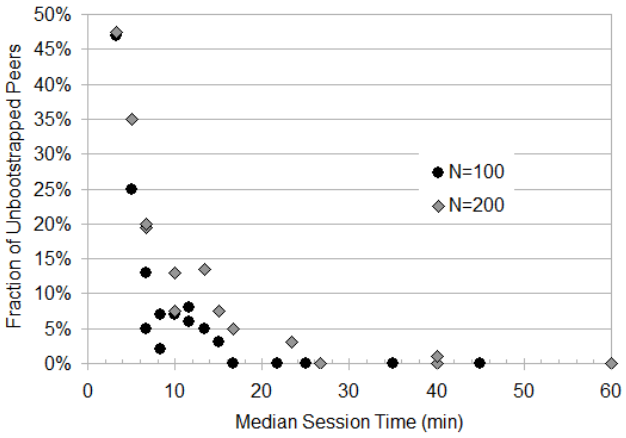

Figure 9: Bootstrap lag rate.

Figure 9 shows the number of peers that are in the process of being bootstrapped as a percentage of the size of the network. This graph exhibits inverse relationship to lookup consistency from Figure 8. If the number of unbootstrapped nodes is high, this means that many nodes are not available for participation.

5 Related and Future Works

In general, DHTs do not behave ideally under high churn. In [1], an attempt is made to study factors that may reduce churn. It compares different methods of choosing peers that participate in DHT.

Performance of Kademlia had been previously studied. In [9], an implementation called Kad was used. Kad is the DHT for the popular P2P network eMule. The paper investigated lookup consistency and performance, using different values for parameters α and K.

Vivaldi, a network coordinate system, was built by [2] on top of Azureus, which also uses Kademlia for its DHT functionality. At high

churn rate, convergence and stability of the coordinate learning algorithm became problematic. It partially worked around the churn problem by reusing learning results from earlier sessions rather than starting from scratch upon a restart. Using Vivaldi was found to speed up network communication as compared to using the XOR distance function.

A different performance measure was studied in [3], whereby the cost of churn is quantified in terms of the amount of data needed to keep the state of a peer current. When nodes transmit more information between each other, lookups can be more successful and occur more quickly. However, there is a rapidly diminishing rate of return.

Most of the experiments in this paper are small scale, simulating multiple instances on a single host. A single host limits the number of concurrent instances of Mojito that can be run. This setup also does not accurately reflect communication costs. The experiment framework, however, is capable of running over a network on multiple hosts, as demonstrated by producing results for Figure 4. Future work can investigate the effect of network latency, and use much larger values for N. These tests can, for example, be set up on PlanetLab servers.

6 Conclusion

This paper conducted experiments to explore the performance under churn of Mojito, which is an implementation of Kademlia that provides DHT capability to LimeWire. The experiments show obvious signs of performance degradation when median session times of nodes dip below 20 minutes, even under fairly ideal conditions without accounting for network communication overhead. Such overhead requires even higher session times to achieve the same performance results.

Therefore, the uptime threshold of 2 hours before a node is allowed to become active in Mojito is necessary, even though it prevents a large number of nodes from fully participating in DHT.

One possible improvement to Mojito is that when a key is associated with multiple values, at least one value should be returned in addition to the secondary keys. Although this provides an incomplete result set, a client may be satisfied with one result, and therefore avoid the cost of performing additional secondary lookups.

References

[1] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. In *Proceedings of ACM SIGCOMM*, pages 147–158, 2006.

[2] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network coordinates in the wild. In *Proceedings of NSDI*, 2007.

[3] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and M. Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of IPTPS*, pages 87–99, 2004.

[4] LimeWire. The Mojito DHT. `http://wiki.limewire.org/index.php?title=Mojito`, 2009.

[5] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS*, pages 53–65, 2002.

[6] Sylvia Ratnasamy, Paul Francis, Scott Shenker, and Mark Handley. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, 2001.

[7] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[8] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of MMCN*, 2002.

[9] Daniel Stutzbach and Reza Rejaie. Improving lookup performance over a widely-deployed dht. In *Proceedings of INFOCOM*, 2006.

[10] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, 2006.

[11] Di Wu, Ye Tian, and Kam-Wing Ng. Analytical study on improving DHT lookup performance under churn. In *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 249–258. IEEE Computer Society, 2006.