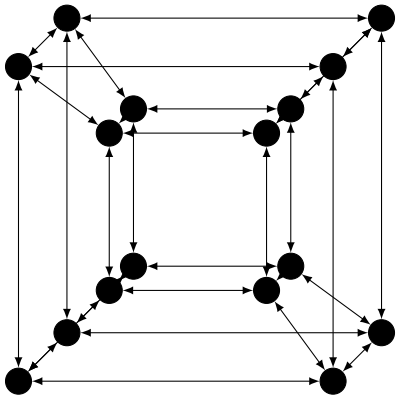


IMPROVING A HYPERCUBE-STRUCTURED DISTRIBUTED HASH TABLE

December 8, 2016

James Wilburn
Poolesville High School
Mentor: Dr. Udaya Shankar
University of Maryland College Park
Department of Computer Science

Abstract

This project modified a preexisting hypercube-structured distributed hash table to improve performance and reduce latency in requests while maintaining the core properties of the algorithm. These benefits were realized by modifying the base algorithm and implementing the replication of hot data to local nodes and the paging of infrequently accessed data. The replication of frequently accessed data reduced the distance requests have to travel, thereby reducing the overall latency per request and increasing the number of requests each peer can serve. The paging of infrequently accessed data reduced memory footprint at a minimal cost to overhead. These modifications addressed inherent limitations of scale in the practicality of distributed hash tables in general. Simulation of the proposed modifications with varying configurations, thread count, and proportion of malicious peers verified that these improvements preserved correctness of the algorithm while enhancing functionality.

Introduction

Peer-to-peer systems are a popular method of providing relatively high-performance file and data storage with minimal investment. As such, numerous peer-to-peer overlays have been developed (e.g. CAN [5], Chord [8], Tapestry [9], Pastry [6], PeerCube [1] to name a few). These systems are all dependent on distributed hash tables (DHTs), consisting of an ID space that is partitioned across multiple peers. Overlays are built to impose various properties on a DHT. Overlays can be structured or unstructured, but structured overlays provide better scalability, efficiency and fault tolerance at the cost of more maintenance [3]. This efficiency and fault tolerance is significantly reduced under high rates of churn. PeerCube provides resistance to a large number of peers leaving and joining the network as well as some resistance to malicious peers [1]. These are the core, desirable properties of PeerCube that are maintained throughout this project.

This paper provides some modifications that can be made to PeerCube to improve performance for frequently accessed data. As a modification of PeerCube, this overlay is also based on a hypercubic topology with clustering at each vertex of the hypercube. The clusters are relatively unstructured. This combination of a small-scale unstructured DHT and a large-scale structured DHT overcomes some of the limitations of each. Firstly, the unstructured clusters never expand to a large size. This ensures that the clusters do not suffer from the same scalability constraints that plague larger unstructured DHTs [4]. Secondly, the clusters are organized hierarchically, with a limited number of core peers that handle traffic and a number of hot-swappable spares. This limits the effects of churn. A new peer does not cause data reshuffling until it is necessary to restructure in order to maintain integrity.

Distributed hash tables in general tend to have relatively poor performance with regard to latency. This is due to each requests requiring a message to be forwarded between many

computers.

The modifications provided are namely the replication of frequently accessed data and the paging of infrequently accessed data. The data is replicated closer to the origin of the request to minimize latency by placing a limited connection that preempts the structure of the DHT. As for the paging of the data, the data is simply stored to disk when it has not been accessed within a specified time-frame. This saves memory at each node at the cost of higher initial latency for infrequently accessed data.

The effectiveness of the modifications were evaluated in a simulation. The simulation mimics a real-world implementation but with mock network transactions. Different combinations of configuration, malicious peers, and processing threads were evaluated to determine the relationship between these variables to latency and performance.

Materials and Methods

The DHT investigated in this paper is based upon PeerCube [1] and as such is composed of hypercube of unstructured clusters. The algorithm as implemented and modified is presented in the following section*. Modifications for replication and paging are highlighted yellow and blue respectively in the following figures.

Algorithm Description

The network exhibits a hypercubic topology. A d-hypercube has 2^d vertices with d edges between each vertex. The hypercube in figure 1 is 4 dimensional. It has 16 vertices with 4 edges on each vertex. The maximum distance, given by the shortest path along the edges, between any two nodes on the hypercube is 4. Hypercubes were chosen for a few of their key properties.

Property 1 (Recursive Construction [7]). *A d-hypercube can be constructed out of lower dimensional hypercubes.*

A d-hypercube is constructed by joining each corresponding vertex of two $(d-1)$-hypercubes. In figure 1, the 4-hypercube is comprised of two cubes such that if each vertex on one cube is given a binary label of 0000 through 0111, the vertex is linked to the vertex on the other cube labeled 1000 through 1111. This means that each vertex is linked only to nodes that differ by exactly one bit i.e. their Hamming distance $\mathcal{H}(a,b)$ equals 1. Property 1 also implies that a hypercube can be dynamically grown and shrunk, effectively changing the dimensionality of the network. This is important in handling the acquisition of new peers and the loss of old peers as it allows the network to change size and structure to reflect the total number of peers.

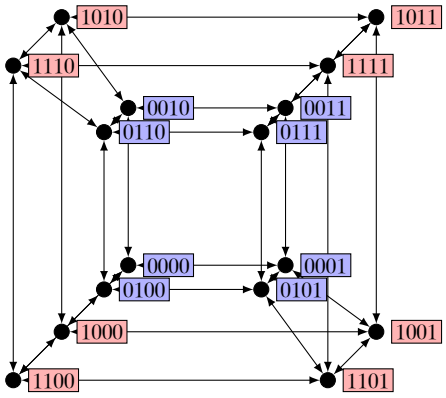


Figure 1: Recursive Construction of a 4-Hypercube

Property 2 (Independent Routes [7]). *If n and m are two vertices on a d-hypercube, there are d independent paths between n and m with lengths less than or equal to $\mathcal{H}(m,n)+2$.*

Independent paths do not share any vertices except the source vertex and the destination vertex. Each leg of the path between n and m changes just one bit. There are $\mathcal{H}(m,n)$ optimal paths of length $\mathcal{H}(m,n)$ and a total of d paths, with a maximum length of $\mathcal{H}(m,n)+2$. In figure 2, the 4-Hypercube has 4 paths and $\mathcal{H}(m,n)=1$ As such, there is one optimal path, colored green, and 3 sub-optimal paths, colored red, purple, and blue.

Clusters

A cluster is a structure containing sets of peers (as defined in section*). Each new peer is randomly assigned a unique random identifier from an m-bit ID space. Random ID assignment prevents maliciously targeted insertion of peers. Groups of peers sharing

Cluster $\mathcal{C}$

Attributes

d: Dimensionality of the cluster

label: A d-bit prefix of the IDs of the peers contained in the cluster

$\mathbf{V_C}$:The set of *core* peers. $|V_C| = S_{min}$

$\mathbf{V_S}$:The set of *spare* peers.

$\mathbf{V_T}$:The set of *temporary* peers.

Figure 3: Cluster Attribute Overview

a common prefix gather in *clusters*. Each cluster is labeled with the *common prefix* of it's contained peers. A cluster's label describes its location in the hypercube and determines which other clusters are its neighbors.

Property 3 (Non-Inclusion)**.** *If a cluster $\mathcal{C}$ exists and is labeled $b_0...b_{d-1}$, then no cluster $\mathcal{C}'$ with $\mathcal{C} \neq \mathcal{C}'$ whose label is prefixed by $b_0...b_{d-1}$ exists.*

The length of a cluster label is the *dimension* of that cluster. A cluster of dimension d is a referred to as a d-cluster and has a label d bits long. The peers in a d cluster maintain a routing table with the d closest neighbors, as defined by distance function $\mathcal{D}$ in equation 1.

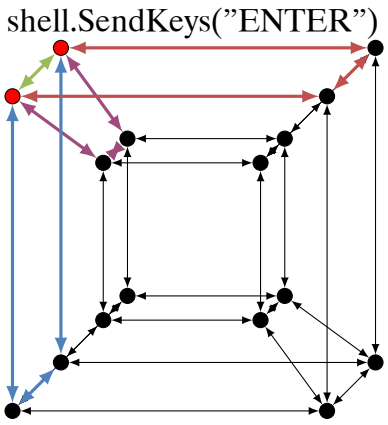


Figure 2: Independent Routes in a 4-Hypercube

A key referring to some data is routed to the cluster whose label is closest to that of the key. Each peer in a cluster is responsible for the same data and data keys. In effect, the data is replicated across all peers in a cluster. Clusters have a maximum size S_{max} and a minimum size S_{min}. S_{min} and S_{max} are constants defined by the probability of peer failure and the proportion of malicious peers respectively.

Topology

Clusters organize into hypercubes by their labels. Figure 4 depicts four 2-clusters with varying numbers of peers organized into a 2-hypercube. An ideal d-hypercube is comprised of 2^d d-clusters. Each cluster is linked to at most d other close clusters, as defined by distance $\mathcal{D}$.

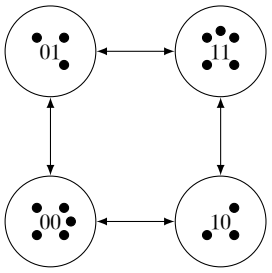


Figure 4: Clusters organized in a 2-hypercube

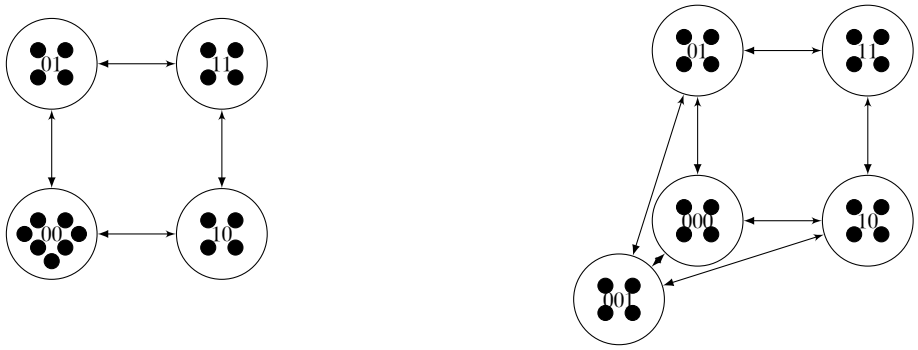
Definition 1. *Let $\mathcal{C}$ and $\mathcal{C}'$ be clusters of dimensionality d and d' respectively. Distance $\mathcal{D}$ can be defined as such:*

$$\mathcal{D}(\mathcal{C}, \mathcal{C}') = \mathcal{D}(a_0...a_{d-1}, b_0...b_{d'-1}) = \sum_{i=0, a_i \neq b_i}^{m-1} 2^{m-i} \tag{1}$$

This function is defined as such so that for any given ID $a_0...a_{d-1}$ and any given distance Δ, there exists exactly one ID $b_0...b_{d-1}$ such that $\mathcal{D}(a_0...a_{d-1}, b_0...b_{d-1}) = \Delta$. This ensures that unique closeness can be determined by each cluster.

Splitting and Merging

Maintaining an ideal hypercube is not always a possible. Due to randomness of ID assignment and churn, clusters may grow or shrink non-uniformly. When the size of cluster $\mathcal{C}$ is less than S_{min}, $\mathcal{C}$ must merge with other clusters into a single cluster $\mathcal{C}'$ with a size greater than S_{min}. On the other hand, when the size of cluster $\mathcal{C}$ is greater than S_{max}, $\mathcal{C}$ must split into two $(d+1)$-clusters each of size between S_{min} and S_{max}.



(a) Network before cluster split

(b) Network after cluster split

Figure 5: Cluster split process

Figure 5 depicts the *split* process. When cluster 00 in figure 5a reaches size S_{max}, it must split. Figure 5b depicts the network after 00 splits into 000 and 001. Cluster 000 remains in the same effective location as cluster 00. Cluster 001 inherits the links of cluster 00 as well as a link to 000. Data is also partitioned and transfered following a cluster split. Data at 001... stored in cluster 00 would be migrated to cluster 001. Merging follows a similar, yet backwards, process progressing from 5b to 5a.

Hot Data Replication

When the data is partitioned during a split, the responsibility for managing replications is also partitioned. In figure 5, the link representing data at key 001... stored in cluster 00 replicated to 11 would be transfered to cluster 001. During a merge process, the responsibility for managing replications would be consolidated in the new condensed cluster.

Peer

A *peer* is the primary structure of this algorithm. Peers perform operations to react to environmental stimuli and maintain an internal state. There are three types of peers: core, spare, and temporary. Core peers are responsible for routing and data handling. Spare peers may be promoted to core at any time, but are not responsible for anything as

spares. Temporary peers are those peers that did not fit in any particular cluster. They may eventually form a new cluster.

The following description of the functions of peers focuses on core peers. For the sake of brevity, only the modifications to the $lookup$ operation are presented in detail. Other modifications were necessary to maintain a valid state of replication, however these are more minor in nature.

lookup Operation

In the base algorithm, a core peer performs a number of actions upon receiving a $lookup$ request. First, the peer p determines the closest known cluster to the destination key. If p belongs to that cluster, p informs all other core peers in the cluster of the incoming request and send a return message to the cluster from which the request originated. If p does not belong to that cluster, p forwards the message to the next closest cluster and wait for $\frac{S_{min}+1}{3} + 1$ responses before returning a response to the immediate previous cluster.

Figures 8 and 9 describe the request and response process. Figure 8 shows how the requests branch and combine at each peer. A peer at cluster 00 initializes the request by sending messaged destined for cluster 11 to cluster 10. Each peer in cluster 10 then forwards the request to each peer in cluster 11. Each peer in cluster 11 then responds to these requests according to figure 9. Upon reception of $\frac{S_{min}+1}{3} + 1$ similar messages, each peer in 11 sends a response message to each peer in cluster 10. Then, each peer in 10 sends a response to the initializing peer in 00. Once $\frac{S_{min}+1}{3} + 1$ similar responses are

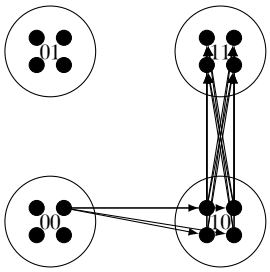


Figure 8: Request

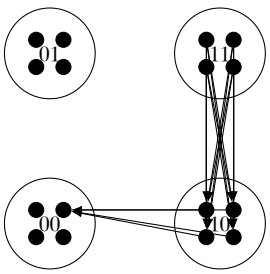


Figure 9: Response to a request

received, the request is complete. All extraneous messages past $\frac{S_{min}+1}{3} + 1$ are discarded.

Replication of Hot Data

Hot data replication is a technique whereby data that is frequently requested is replicated to faster or closer storage to minimize lookup times. A basic algorithm for implementing this is described in [2]. In essence, each peer keeps a registry of which peers requested the data in recent time. If a peer requests data more than T_{cache} times in the time frame $T_{replicate} \cdot \mathcal{H}(p, origin)$, the peer is added to the registry RH_{cache} so that each new update to the data also updates the replicated data. In order to implement this improvement, the *lookup* operation was modified as such (highlighted in figure 7):

The first modification is such that $lookup$ requests record transit history. This is so that the peers in the owner cluster of key can count how many times each cluster has requested key. This count is maintained in the Request Handler for Replication Counting (p.RHRC[key, $\mathcal{C}$.Label]). The second modification makes each peer check a threshold for replication of key to each $cluster$ in $transithistory$. When the threshold $T_{replicate}$ is reached on any element in p.RHRC, p issues a special $replicate$ request to all core members of that cluster.

Peer p also maintains the Replication Mapping Handler (p.RMH) such that p.RMH[$\mathcal{C}$.Label] is the set of keys replicated in $\mathcal{C}$. This state is important for updating the replications on subsequent put requests. When data is replaced via a put request or removed via a $delete$ request, all replicated data stored in other clusters is also replaced or removed. The state information that allows this is stored in p.RHRM.

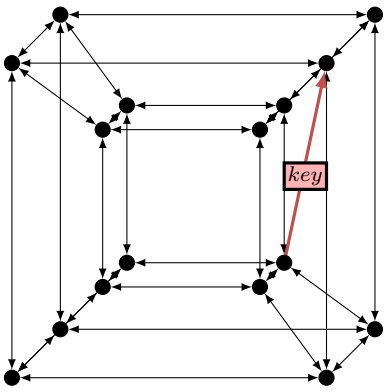


Figure 10: Replication routing

Figure 10 illustrates how the replication system preempts the standard structure of the hypercube. The red line represents a replication of some data at key between two usually unlinked clusters. Any $lookup$ for key that passes through the replicated store short-circuits and return the data for key. This preemption of the standard structure of the DHT allows for a shortened path to the requested data.

Paging

In the base implementation of the algorithm all data is stored in memory. This is fine for small amounts of data, but unsustainable for a larger data-store. Paging would solve this issue by offloading infrequently accessed data to the disk, freeing memory space for more frequently accessed data. This is a simple modification to implement. The choice to offload data to disk can be entirely local as it does not meaningfully affect the state of the network as a whole. Each peer counts how often each piece of data is requested. If a particular datum falls below a threshold T_{page}, it is offloaded to temporary disk space.

Simulation Procedures

The simulation attempts to closely mimic actual network traffic. The boundaries between peers are respected, and all data transfered between peers is copied. The network transactions are simulated asynchronously. The simulation is extremely similar to a real-world implementation of the algorithm, only with all network calls replaced by randomly delayed queues with a chance for failure or timeouts. This chance is randomly defined by constraints $P_{success}$ and $P_{timeout}$, which are constant for this simulation, but may vary in future investigations. The simulation also maintains a global state of the network so that further analysis into the inner workings of the network may be done. Seeing as how this simulation is not a real-world test, only the relative performance of the various configurations is significant. The absolute magnitude is largely irrelevant and almost entirely

dependent on the testing hardware.

Malicious peers are simulated as purely non-functional for the purposes of this simulation. They are not a primary focus of the investigation, and as such, their impacts are largely assumed to be similar to that of a failing node. Further investigations into the behaviors of the DHT with properly malicious nodes of various degrees may be conducted at a later date.

Table 1: Simulation configurations

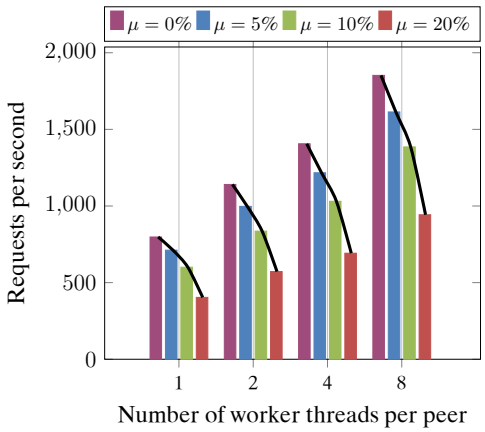
μ	Workers	Configuration
0%	1	Default
5%	2	Caching
10%	4	Paging
20%	8	Caching and Paging

The simulations conducted covered the 64 distinct cases described in table 1. μ is the percentage of malicious peers in the network. Considering the limitations of maintaining this exact percentage while cycling peers in and out of the network, μ was approximated as the chance each new peer has to be malicious. Seeing as peers are randomly chosen to leave the network, this approach maintains μ at approximately the intended value.

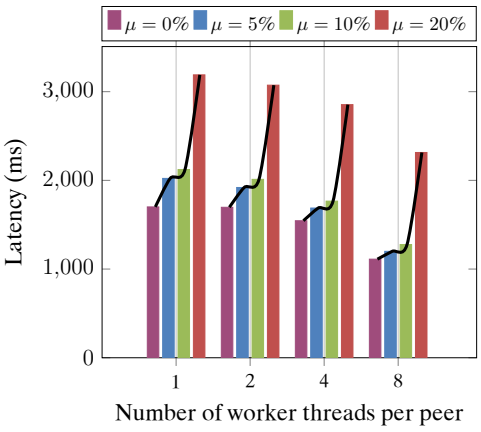
For the purposes of the simulation, peers were then added at a semi-constant rate of about $1\frac{peers}{second}$. Roughly 1% of all peers were removed each 10 second period. Data collection began once the network reached an equilibrium. A random sequence of put and $lookup$ requests were sent to random nodes in the network. Every minute for 30 minutes, the network was paused and data on requests per second and 95th percentile latency were collected. 95th percentile latency was chosen to measure relative worst-case performance while not being too susceptible to temporary spikes in latency. After the 10 minutes, the results of each minute of observation were averaged.

Results

Graph 1: Mean Requests per Second (Unmodified)

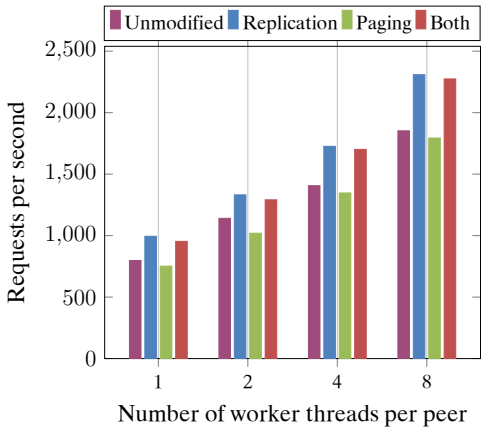


Graph 1: Mean 95th Percentile Latency (Unmodified)

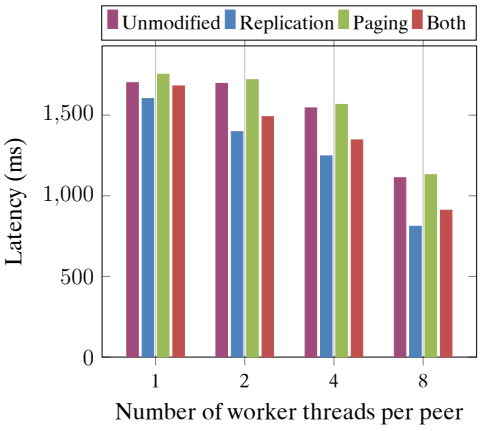


Graphs 1 and 1 present the mean of 10 minutes of data collected for each of 16 simulations. Graph 1 is the mean number of requests per second on the base configuration. It shows decreases with respect to μ, the percentage of malicious peers, and an increase with respect to the number of worker threads per peer. Graph 1 is the mean of the 95th percentile latency measurements collected each minute for 10 minutes on the base configuration. It shows an exponential increase in latency with respect to μ and a decrease in latency with respect to the number of worker threads per peer.

Graph 1: Mean Requests per Second ($\mu = 0$)

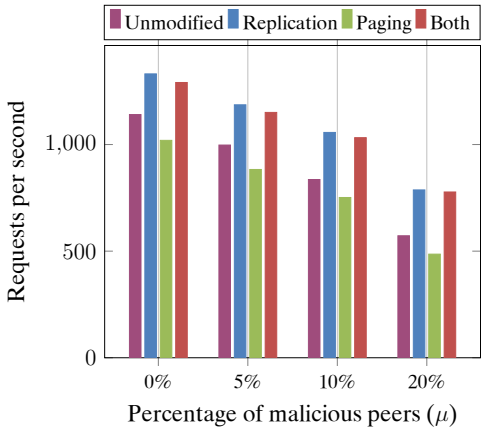


Graph 1: Mean 95th Percentile Latency ($\mu = 0$)

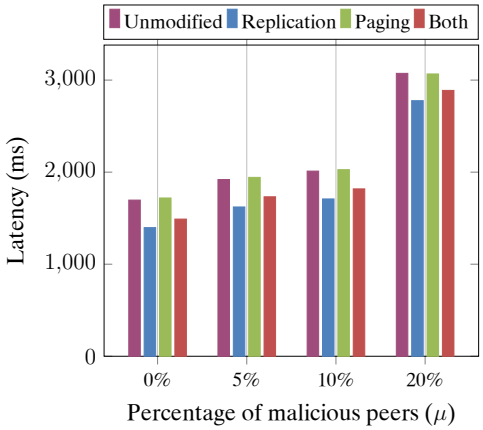


Graphs 1 and 1 show the relative performance of the various configurations of the algorithm with respect to the number of worker threads w. The algorithm with replication showed between 6% and 30% lower latency than the base configuration, depending on the number of worker threads. Replication also demonstrated a 22% increase in the number of requests per second over the base configuration. Paging alone resulted in slight decreases in performance across the board. 95th percentile latency increased by between 1.7% and 3.1% and the number of requests per second decreased by 3.2% to 5.9% from the default configuration. The combined configuration of replication and paging shows between a 1.2% and a 20% decrease in latency and a 18% to 21% increase in the number of requests per second.

Graph 1: Mean Requests per Second ($w = 1$)



Graph 1: Mean 95th Percentile Latency ($w = 1$)



Graphs 1 and 1 displays the relative performance of the various configurations of the algorithm with respect to μ. All of the configurations showed exponential increases in latency with respect to μ. The replication and combined configurations had relatively lower latency and higher performance than the base configuration for all values of μ while the paging configuration had relatively higher latency and slightly lower performance.

Discussion

The results of the investigation support the objectives of the project: to improve the performance of a particular DHT while maintaining some key characteristics. Reference [1] presents models for the probability of request success of PeerCube as it relates to μ. These models project an exponential decay in the probability of request success. The exponential increase of the latency with respect to μ, as shown in graph 1, is consistent with the previous PeerCube simulation findings. The previous finding are within the results of the simulation (with 95% confidence when compared on a normalized scale of 0-1). This would imply that the algorithm was implemented correctly in the simulation.

The inconsistent relative change with respect to the number of worker threads per peer in graphs 1 and 1 can explained as issues with single-threadedness. With just one thread, thread safe operations, such as $lookup$, cannot run in parallel. This means that every $lookup$ pauses the network for the length of the request. This would also imply that the overhead of supporting operations ($lookupreturn$,$lookupinform$,$replicate$) is significant (non-negligible, ($p<0.05$)), but mitigated by parallel processing. Together, these two factors explain why the single threaded replication configuration performed relatively worse than expected compared to the multithreaded replication configurations.

The overall performance of the replication configurations is unsurprising. Data requested from similar $lookup$ operations are replicated closer to the requester, so latency would tend to be much lower. Also, with lower latency and travel distance, fewer peers spend less time waiting for responses, so the peers have more time on average to serve other requests. And so requests per second would be higher on average. The results support that these differences in latency and requests per second are significant ($p<0.05$).

The relative performance of the paging configurations was also as expected. Writing and reading from hard-disks, even simulated ones, takes time and CPU time. As

such, a paging peer would perform more slowly, yet use less memory to store the same amount of data. The results of the combined approach were similarly expected. Replication had a more significant positive effect on performance than the negative effect of paging ($\Delta rps_{rep} > -\Delta rps_{page}$, $\Delta latency_{rep} > -\Delta latency_{page}$, p<0.05). As such, the combined approach performed strictly better than the base configuration.

The results of the simulations with respect to the number of worker threads was somewhat unexpected. An increase in the number of requests per second was expected, but this increase was expected to be somewhat higher. The increase was significantly less than linear with respect to the number of threads ($R^2 = 0.3$). This may have been due to all of the non-threadsafe operations pausing the worker threads. Further investigation with granular measurements of operation type and time would be required to determine if this is the case. This may also have been due to physical processing constraints in the simulation environment. The execution of thousands of green threads may have reached the limits of the testing environment. Further testing on more robust hardware may be necessary.

All of the simulation configurations performed appropriately with respect to μ. Each configuration experiences slow exponential growth in latency with respect to μ. This is not a hugely positive outcome, but this still suggests that most requests completed successfully.

Overall, the investigation validated that hot data replication is applicable to peer-to-peer overlays and effective in reducing latency. The investigation also demonstrated that the paging of data results in relatively insignificant performance degradation for a relatively larger decrease in memory usage.

Peer *p*

Attributes

cluster: The *cluster* to which p belongs

ID: An m-bit unique identifier. p.ID is prefixed by p.cluster.Label. Notation: $p.\text{id}_0 ... p.\text{id}_i ... p.\text{id}_{m-1}$

RT: (Routing Table) An array of clusters such that the cluster in p.RT[i] has a label that differs in bit i

RHLC: (Request Handler for Lookup Count) A map of IDs to integers; RHLC[key] returns the number of pending *lookup* requests for key

Upon: reception of $(\frac{S_{min}+1}{3}+1)$ ($'lookup'$, key, q, $origin$, $travelhistory$) messages from the network with the same key and $origin$
Input: p as the local peer

*

References

[1] E. Anceaume, R. Ludinard, A. Ravoaja, and F. Brasileiro, "Peercube: A hypercube-based p2p overlay robust against collusion and churn," in *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 2008, pp. 15–24.

[2] R. Frank, G. Arun, R. Anderson, R. Mediouni, and S. Klein, *Method and Apparatus for Selective Data Caching Implemented with Noncacheable and Cacheable Data for Improved Cache Performance in a Computer Networking System*. Google Patents, Feb. 2000, uS Patent 6,021,470.

[3] T. Locher, S. Schmid, and R. Wattenhofer, "Equus: A provably robust and locality-aware peer-to-peer system," in *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*. IEEE, 2006, pp. 3–11.

[4] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys & Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content-Addressable Network*. ACM, 2001, vol. 31, no. 4.

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[7] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Transactions on computers*, vol. 37, no. 7, pp. 867–872, 1988.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[9] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph, and others, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," 2001.