The P2P MultiRouter: a Black Box Approach to Run-time Adaptivity for P2P DHTs*

James Newell and Indranil Gupta
Department of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL 61801.
{jnewell2, indy}@cs.uiuc.edu

Abstract

Peer-to-peer distributed hash tables (p2p DHTs) are individually built by their designers with specific performance goals in mind. However, no individual DHT can satisfy an application that requires a "best of all worlds" performance, viz., adaptive behavior at run-time. We propose the *MultiRouter*, a framework that works with multiple DHT protocols running at a node. The MultiRouter interacts with DHTs in a black box manner, i.e., through well-defined interfaces. We study the design and performance of a MultiRouter instance that uses simple cost functions and adaptive choices to provide a best-of-all-DHTs run-time performance with respect to object access times and churn-resistance.

1 Introduction

Peer-to-peer distributed hash tables (p2p DHTs) are built to satisfy specific, but differing, individual goals of performance. On the other hand, a p2p application typically requires a "best of all worlds" performance at run-time. The result is that no single DHT turns out to be just "right" for that application's requirements.

For instance, a specific p2p application might require fast access to objects, even immediately after the object is inserted into the p2p system. Constant lookup time DHTs (e.g., [7] or [6]) require some time to replicate meta-data, and can thus offer fast access only a while after object insertion. Other DHTs (e.g., [12]) can guarantee logarithmic lookup times right after object insertion, but may be cumbersome to tune for constant time lookups. Another example is churn-resistance: DHTs exhibit differing behavior under the same churn [10], while a p2p application cares only for a best of all worlds churn-resistance.

Some incremental approaches to this problem have augmented DHTs with extra or exogenous links [9]. Others define an underlay [11] such as a ring [4] that can be used to "tie" together DHTs that are built using that common underlay. However, this is too restrictive; many existing DHTs such as Kelips and CAN do not use a ring at all.

Our approach to the problem is motivated by the simple fact that p2p DHTs consume very small amounts of resources (memory, bandwidth) at each node. This makes it feasible to run multiple DHTs at some (or perhaps each) node in the system. Objects can be inserted into one or more of these DHTs, and fetched in parallel, thus offering the "best of all worlds" performance to an application.

While some readers might view this approach as simplistic, in fact, the Internet uses similar techniques to solve a related problem. Multihoming in the Internet ties together different types of LANs [1, 2]. Internet bridges and routers connect subnetwork protocols.

Our solution consists of the *MultiRouter*, a protocol that accesses multiple DHTs in a black-box manner, i.e., through a a well-defined interface of {*insert*(*object*), *lookup*(*object*)}. This relation is depicted in Figure 1. The MultiRouter framework can be realized by plugging in several run-time estimators and cost functions

*This research was partly supported by National Science Foundation Grant ITR-0427089

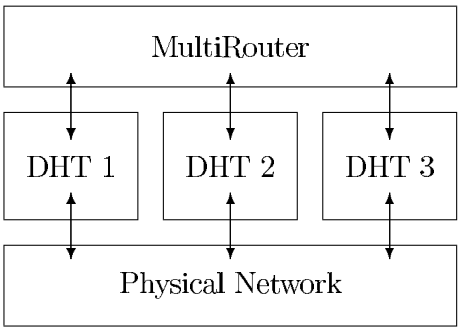

Figure 1: MultiRouting Framework with uniform insert/lookup interface

that determine the right choice of which DHT to use on the fly. As a proof of concept, our particular choice of cost functions in this paper is directed at only two specific application requirements: object lookup times, and churn-resistance.

Seen another way, the MultiRouter spans an "over-overlay" that logically sits on top of multiple, arbitrary overlays. Below, we describe the MultiRouter framework, the currently implemented functionalities, and describe data from trace-driven experiments on a prototype implementation.

2 Design Description

The MultiRouting architecture consists of one or more underlying DHT substrates communicating with the overarching MultiRouter overlay (Figure 1). The user application can now interface through the MultiRouter {*insert*(*object*), *lookup*(*object*)} API instead of with each individual DHT. The DHTs will run concurrently and independently of one another with direct access to the physical network. The user can customize her MultiRouter to the surrounding networks by "plugging-in" any combination of DHTs that are available.

When the MultiRouter initially starts, it will order all its substrates to run its implementation-specific join protocol to connect each DHT to its corresponding logical network. Once they are all finished connecting, the MultiRouter is available to perform object operations such as *insert* and *lookup*.

During an insert, the MultiRouter will send this key in parallel to all its underlying DHTs. The various DHT implementations will select different targets to house the newly inserted key. It should be evident that by inserting the object on many overlays, the MultiRouter achieves a greater random distribution of replicas and therefore a better probability of success under harsh network conditions.

Once the insertions have completed, the MultiRouter will add an entry about the object in its *StatsTable*. The purpose of the *StatsTable* is to gather and maintain specific metrics for each object it knows about. The type and weight of such metrics can be customizable to the application using the MultiRouter.

When a user requests to lookup a key, the MultiRouter uses the metrics provided by the *StatsTable* to determine which DHT(s) to route the request on. These metrics are entered into a cost function $C_{DHT}(x)$ that calculates the relative score of that substrate for that particular file. The MultiRouter will use this score as an indicator for how well the DHT has been performing compared to other DHT(s). Given a set of rules, the MultiRouter will pass the lookup to the DHTs it decides will probably perform the best for this request.

Formally, given an array of statistical data d_i, we define M to be a set of metric functions $(m_1, m_2, \ldots, m_k \in M)$, where $m_i(d_i, t) \rightarrow \mathcal{R}$. The cost function is then given by:

$$C_i(M) = \alpha_1(m_1) + \alpha_2(m_1) + \ldots + \alpha_k(m_k)$$

Where the sum of α's is 1. Given j DHT substrates, we have a set of j cost functions $\mathcal{C}$, where $(C_1, C_2, \ldots, C_j \in \mathcal{C})$. To interpret these costs, we define a set of rules using a rule function $R(\mathcal{C}, C_i) \rightarrow Bool$. The union of these rules provides a set of booleans that determines which DHT(s) the query should be routed on. When a lookup from DHT_i is eventually returned to the requester, it will update d_i to reflect the new metrics obtained from the recent query, and pass along the result up to the application.

3 Implementation

We developed a fully functional MultiRouter prototype using basic parameters common to most applications. To evaluate this prototype a discrete-event, link-layer simulator, and two DHT substrates, Kelips and Pastry, were also implemented. In addition, to create a uniform interface between the MultiRouter and

the underlying substrates, a simple filetuple $\langle key, NodeID \rangle$ storage system similar to Kelips was implemented between Pastry and the MultiRouter. All of the code was written in Java 1.4.2.

A *join* request on the MultiRouter prototype will initiate a join on both Kelips and Pastry and reset the filetuple metrics data. Once all the substrates indicate they are ready for operation, the MultiRouter will accept requests and inserts from the application.

By default, our MultiRouter implementation inserts a filetuple on all DHT substrates. While in reality this is not necessary for all files, it does help show the strength of the MultiRouter design. A filetuple lookup will initiate the prototype's cost function and rule set to determine which substrates to route the query on. Our prototype implementation uses two metrics to gauge DHT performance: latency and failure rate. These two metrics were chosen because they are common to most applications. They also encompass a relative long-term and recent performance history respectively. To smooth out the fluctuations of varying latency, our prototype employs an aging function. Given a new latency datum $l \in [0,1]$ (where 1 is proportional to a *timeout*) and $\beta \in [0,1]$:

$$latency_i(l) = \beta * l + (1 - \beta) * latency_{i-1}$$

While β can be varied for different behaviors, we fixed $\beta = 0.05$ in our implementation. Therefore, a single latency spike will not affect the cost function greatly.

For the failure rate, our prototype uses a decaying function dependent on the current time, t. Given a new failure rate datum $f \in \{0,1\}$, where $f = 1$ upon a returned failure or *timeout* and 0 otherwise and $\gamma \in [0,1]$:

$$failure_i(f, \Delta t) = f + \gamma * failure_{i-1}$$

Since this function is run on every round, Δt, even if no new failure had been detected (in which case $f = 0$), the failure metric will eventually approach zero depending on the value of γ. In our prototype, we fixed $\gamma = 0.75$ and $\Delta t = 1$ sec. Note that recent failures will always trump a low latency history; however, this dominance does not persist long due to the decaying component of the function. This will prevent a transient failures from having a long-term effect on the cost function. Ideally, the MultiRouter can now select an alternate and maybe less preferred DHT when the primary DHT is failing. If the failures are temporary, the MultiRouter will be able to "switch back" to the preferred DHT in a moderate amount of time.

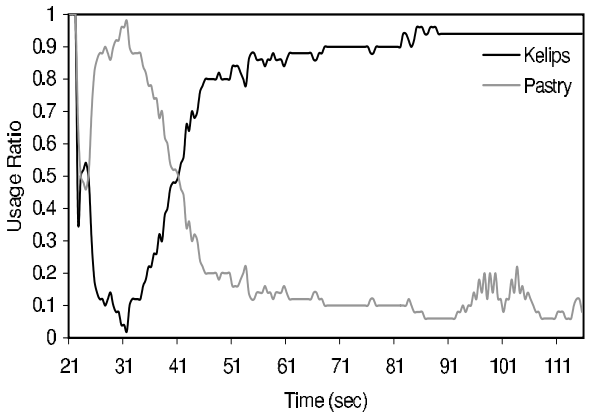

Figure 2: MultiRouter Adaptivity: Right after an object is inserted, the MultiRouter prefers the Pastry for lookups, since it is faster. However, after the object meta-data has been replicated enough for the Kelips substrate to have better lookup times, the MultiRouter switches over.

Given these two metrics we can compute the cost function for a given DHT. Derived from the equation in Section 2, the cost function for our implementation is

$$C(\mathcal{M}) = \alpha_1(m_1) + \alpha_2(m_1)$$

Since both metrics are equally important, we set $\alpha_1 = \alpha_2 = 0.5$ in our prototype.

The way the costs are used is defined by the implementation's rule function R. Our prototype's rule function is defined as follows:

$$R(\mathcal{C}, C_i,) \quad = \quad \begin{cases} \text{true} & cold \\ \text{true} & retry > 1 \\ \text{true} & \min(\mathcal{C}) > C_{threshold} \\ \text{true} & \min(\mathcal{C}) = C_i \\ \text{false} & otherwise \end{cases}$$

We considered a DHT cold when the filetuple has been recently added or the DHT has not been used for a certain threshold. A DHT will not be considered warm until it is included with three consecutive queries. This gives the MultiRouter a base for newly inserted files and allows performance checks after a certain interval.

We also allow the MultiRouter to query all DHTs on a retry. This is because under failure, it is important to determine in an efficient manner which DHT has the filetuple available. Furthermore, if the minimum cost is greater than

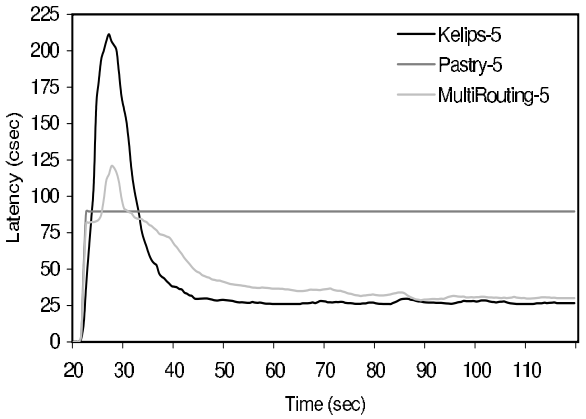

Figure 3: 5 sec-Avg of Lookup Latency of Micro-benchmark: Immediately after insertion, Kelips has a much higher lookup latency than Pastry, whereas later the situation reverses. The MultiRouter latency is close to the best performing DHT throughout the experiment.

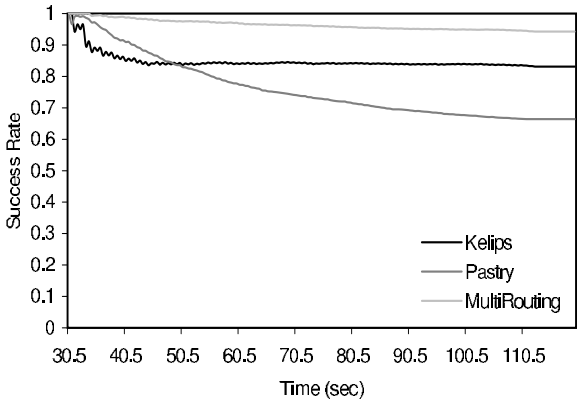

Figure 4: Effect of Generic Churn: The MultiRouter is able to guarantee lookup success rates better than either Kelips or Pastry when there is churn. This is due to replicated files and adaptive lookups.

some $C_{threshold}$ then we know all networks are performing poorly. In this case, there is a significant probability that the initial query will fail. Therefore, to increase the chances of success, the MultiRouter will query all substrates simultaneously.

4 Experimental Results

Using the simulator and prototype described in Section 3, we evaluated a MultiRouter implementation using simulation runs and trace-based experiments. The experiments were run on a 3.0 GHz WinXP P4 with 512 MB of RAM.

4.1 Micro-Benchmark

The micro-benchmark uses a trans-stub topology of 200 nodes using the *GT-ITM* [1] generator. After the network stabilized, we inserted a filetuple into both substrates and selected a node at random to continuously query this sole filetuple every 0.5 sec for 2 minutes.

Figure 2 shows the DHT usage ratio by the MultiRouter. Initially, Kelips will perform poorly due to the replication delay, so the Pastry overlay will be heavily favored. However, after the file is propagated throughout the network, Kelips becomes the new favorite because of its $O(1)$ lookup time.

We also averaged the latencies of the last 5 queries over 50 runs. Figure 3 shows that Kelips does slightly better than the MultiRouter after the large spike. This is because the switch over does not happen instantaneously due to the failure-cost decay function described in Section 3. However, eventually the MultiRouter's performance does converge to the Kelips' latency.

4.2 Generic Churn

Using the same topology as the micro-benchmark, we simulated churn on our simulator by having two members join and leave every second for 120 seconds. During this time we also had a random node query two filetuples every second from a previously inserted set of 100.

The first measure of performance is the success rate of the queries. The results for this experiment are shown on Figure 4. The MultiRouter has the highest average success-rate over 50 runs of this simulation. It has a performance gain of about 10% better than Kelips and 35% better than Pastry.

Figure 5 presents the average latency of the last 5 lookups to determine if performance was sacrificed for an increased success rate. Fortunately, the MultiRouter performs better of the three. It is also evident from the figure that the MultiRouter probably tends to favor the Kelips substrate during most runs, but achieves a boost by using Pastry during periods of poor Kelips performance.

4.3 Overnet Trace

To simulate a more realistic churn scenario, we also ran a similar experiment using availability traces from the Overnet P2P network [3]. We

[1] http://www.cc.gatech.edu/projects/gtitm/

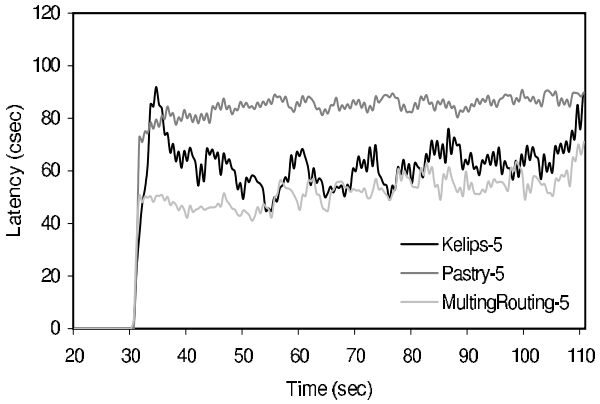

Figure 5: 5 sec-Avg of Lookup Latency for Generic Churn: The MultiRouter achieves a lower latency than Kelips and Pastry by utilizing the better-performing DHT.

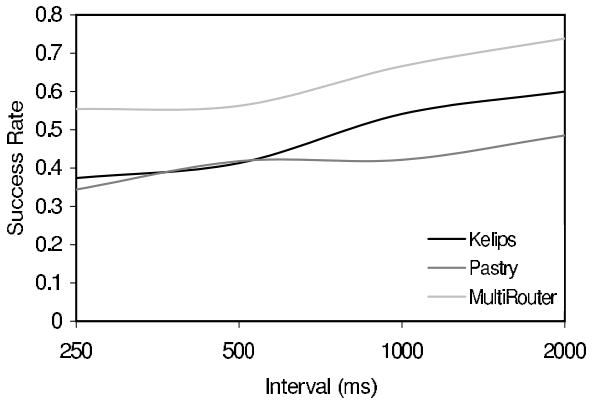

Figure 6: Effect of Varying Trace Interval: The MultiRouter demonstrates higher success rates under varying levels of churn.

mapped 500 of these traces to a 1275 trans-stub topology. The remainder of the nodes remained alive during the entire experiment to maintain 100 inserted filetuples. Similar to the previous experiment we queried two filetuples every 0.5 sec from random nodes for 5 minutes.

To align this trace to the previous experiment we scaled-down the interval between .25 seconds to 2 seconds and calculated the average success rate of each overlay (Figure 6). In general the same relationship from the previous experiment holds between Kelips, Pastry, and the MultiRouter.

Caveats The caveats of the MultiRouter are the increase in resource usage at individual nodes, **i.e.** bandwidth and memory utilization. However, the effect of this drawback can be reduced by choosing the DHT substrates wisely. For instance, the overhead of using Kelips (3.90 msg/sec) and Pastry (0.46 msg/sec) is simply the summation of the component's requirements (4.36 msg/sec). Furthermore, the marginal overhead is not substantial compared to the individual DHT with the highest overhead.

5 Conclusion

The MultiRouter, an over-overlay to the user application, has been shown to improve the success rate and latency of DHT lookups by increasing the availability of inserted objects. By replicating objects on multiple independent substrates, the MultiRouter is capable of taking advantage of the differing properties provided by alternate DHTs. As a result, it has the powerful ability to adapt to changing network conditions by effectively calculating which substrate is most likely to be successful. The MultiRouter is an extension of the concepts of bridging and multihoming widely used in the Internet, to peer-to-peer systems.

References

[1] A. Akella, et al. "A Comparison of Overlay Routing and Multihoming Route Control." In *Proc. of ACM SIGCOMM*, Aug 2004.

[2] A. Akella, et al. "A Measurement-Based Analysis of Multihoming." In *Proc of ACM SIGCOMM*, Aug 2003.

[3] R. Bhagwan, S. Savage, and G. Voelker. "Understanding Availability." In *Proc. of IPTPS*, Feb. 2002.

[4] M. Castro, et al. "One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks." In *Proc. of 10th European SIGOPS Workshop*, Sept 2002.

[5] Clip2. The Gnutella Protocol Specification v0.4. http://www.clip2.com

[6] A. Gupta, B. Liskov, "One Hop Lookups for Peer-to-Peer Overlays", In *Proc. of HotOS*, 2003.

[7] I. Gupta, et al. "Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead." In *Proc. of IPTPS*, Feb 2003.

[8] P. Linga, I. Gupta, and K. Birman. "A Churn-Resistant Peer-to-Peer Web Caching System." In *Proc. of ACM WShop. SSRS*, Oct 2003.

[9] S. Marti, P. Ganesan, and H. Garcia-Molina. "DHT Routing Using Social Links." In *Proc. of IPTPS*, Feb 2004.

[10] J. Li et al, "Comparing the performance of distributed hash tables under churn", In *Proc. of IPTPS*, Feb 2004.

[11] A. Nakao, L. Peterson, and A. Bavier. "A Routing Underlay for Overlay Networks." In *Proc. of ACM SIGCOMM*, Aug 2003.

[12] A. Rowstron and P. Druschel. "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems." In *Proc. IFIP/ACM of Middleware*, Nov 2001.

[13] D. Stutzbach and R. Rejaie. "Towards a Better Understanding of Churn in Peer-to-Peer Networks." Uni. of Oregon, Technical Report CIS-TR-04-06, Nov 2004.