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 light-weight solution that provides adaptivity to the application using a DHT-independent approach. By merely making run-time choices to select from among multiple DHT protocols using simple cost functions, we show the MultiRouter is able to provide a best-of-all-DHTs run-time performance with respect to object access times and churn-resistance. In addition, the MultiRouter is not limited to any particular set of DHT implementations since the interaction occurs in a black box manner, i.e., through well-defined interfaces. We present microbenchmark and trace-driven experiments to show that if one fixes bandwidth at each node, the MultiRouter outperforms the component DHTs.

1. Introduction

There is currently a wide variety of peer-to-peer distributed hash tables (p2p DHTs) [6, 9, 16]. Each is built to satisfy specific, but differing, individual goals of performance. However, p2p applications typically require a "best of all worlds" performance at run-time. As a result, each individual DHT implementation has to independently address the issue of adapting to varying networking conditions. Typically, a mechanism to mitigate the problem is "patched-in" after the DHT was designed, and hence limited to the underlying architecture of the implementation.

For example, to bypass unavailable node or links, a DHT will typically retry an operation using an alternate route or method when faced with failure. Creating alternate routes in DHTs that lack randomness can cause great complexity with limited results. It should be easy to see that it is difficult for a single implementation to continually provide a diverse set of efficient routes to the same destination. Furthermore, every additional attempt typically relies on the same or similar routing techniques for each reroute and hence is susceptible to inherit properties of the routing implementation itself. For instance, constant lookup time DHTs (e.g., [8] or [9]) require some time to replicate meta-data, and can thus offer fast access only a while after object insertion. Therefore, they usually lack an ability to locate objects that have been recently inserted.

In addition to failure, DHTs need to handle high-levels of churn. It is well known that the average lifespan of peers in p2p networks is dismally low, which results in a high level of churn in the system [15, 17]. As a result, many problems such as missing files or stale routing tables can easily occur. To overcome this, a majority of the implementations employ an object replication mechanism to increase the availability of the object. Typically, replica discovery is a DHT feature that is added in after-the-fact. As a result, the primary strength of DHT routing lies in locating "primary" DHT objects and not replicas.

We propose an alternative solution to the problem of adaptivity that works by inserting a totally new layer between the application and the DHT routing mechanism. This new *MultiRouter* layer is a simple and light-weight overlay that lies on top of one or more independent, underlying DHT substrates. By effectively calculating which substrate(s) is most likely to be successful by using several run-time estimators and cost functions, the MultiRouter provides the user-application the powerful ability to adapt to changing network conditions by exploiting the differing availability properties provided by disparate routing architectures and using the strength of each individual DHT of locating its "primary" copy of a DHT object. Furthermore,

*This research was partly supported by National Science Foundation CAREER Grant CNS-0448246.

since the MultiRouter only communicates to the unmodified DHT implementations using a well-defined interface of $\{insert(object),\ lookup(object)\}$, the MultiRouter can easily provide a "best of all worlds" performance that is *DHT-independent*.

At first glance it may appear that the idea of a MultiRouter would greatly increase the complexity of any system due to the execution of multiple DHT implementations that run in parallel. However, the strength of the MultiRouter architecture lies in its simplicity. By effectively selecting only the best implementation to use for any given query, the complexity of the system is limited to just the best performing DHT(s). This is because the less performing and seemingly more complex DHTs will be left unused, and hence do not add to the complexity of the system. Furthermore, the MultiRouter layer itself is very light-weight and adds little additional complexity to the overall system. As a result, the MultiRouter can be considered a simple solution to obtain adaptive behavior and avoid inflexible complexity.

We show that the MultiRouter is capable of improving both the success rate and the latency of DHT lookups by using experimental churn data from simulations and Overnet data traces. We also exhibit the adaptability properties of the MultiRouting technique that allow the MultiRouter to easily handle transient DHT problems. Finally, the additional message overhead of the MultiRouter is analyzed and shown not to have a substantial effect.

Roadmap: Related work done in this area is presented in Section 2. Section 3 describes the overall design of the MultiRouter architecture. Our implementation prototype is given in detail in Section 4. Section 5 discusses the experimental evaluation of this prototype. Finally, future work and the conclusion are given in Section 6.

2. Previous Work

While this approach has never been directly researched before, there has been some interesting work done that is relevant to particular aspects of the project. For example, there has been some cursory work of using adaptive concurrent overlays, but not to the extent we wish to pursue. One notable example [13] is the use of exogenous social networks as an alternative overlay on top of an underlying DHT (such as Chord). While the primary focus on this paper is the reliability of message forwarding using trust, it encapsulates many of the same ideas as MultiRouting. Another paper [12] also proposes the use of concurrent DHTs. However, the scope of the paper is focused on file-sharing and assisting key-word based searches. Specifically, they advocate assisting the Gnutella P2P network by implementing an auxillery but parallel DHT with differing search properties to support the less popular and more difficult to find files.

Two other papers [5, 14] also propose the use of multiple overlays. However, their main motivation is to provide a universal underlayer to serve many competing and independent overlays. However, the two papers deviate about its purpose. One provides a service for bootstrapping incoming nodes to locate a particular implementation, while the other focuses on caching similar functions used by the overlays. This prevents redundant work from being done. Neither of them makes the assumption that nodes will use multiple overlays at the same time on underlayer substrate.

Multihoming [1, 2, 7] and Resilient Overlay Networks (RON) [3] are other designs that have comparable characteristics to MultiRouting. In multihoming, the node is an end-user who is connected to multiple ISPs. The multihomed user now can make a decision about which ISP to request services from. This allows the user to choose a route to minimize link latency, maintain load-balancing, and provide fault-tolerance. These are the same kind of choices that occur on MultiRouters; except in our case, nodes will be choosing which DHT to route on instead of an ISP.

RONs allow users to quickly reroute around BGP link failure using a third-member as a relay even though it has little information about the underlying topology. This idea is similar to MultiRouting because each member of a RON needs to decide if it should route directly using the physical network or attempt to route via another RON member. MultiRouters make the same type of choices using similar methods.

3. 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. Since the DHTs are used as is, MultiRouters are compatible with traditional DHTs networks.

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. For example, Pastry will choose the node with the most similar ID to the object key, whereas Kelips will

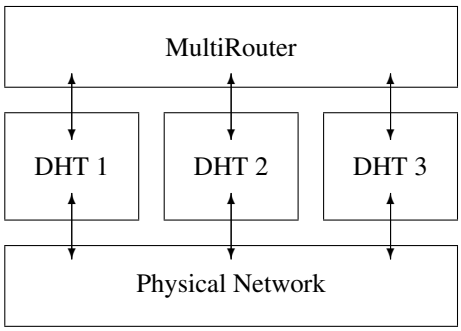


Figure 1. MultiRouting Framework with uniform insert/lookup interface

randomly select a home from a group of nodes determined by a hash of the key. Furthermore, these objects are replicated using differing techniques for each individual DHT implementation. Because the MultiRouter is not limited to a single DHT's algorithm, it should be evident that the MultiRouter achieves a greater available distribution of replicas by inserting the object on many overlays. As a result, the probability of a successful lookup increases when network conditions become harsh with little sacrifice of performance.

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. For example, a DHT implementation of DNS may emphasize low-latency lookups whereas a cooperative caching application may value freshness of content (Table 1). However, it should be noted that the specific impact that these various combinations of metrics and weights can have on the MultiRouter's performance is still an open issue that we hope to address with future research. Initially, all statistics about a newly inserted file will be cold, but a joining MultiRouter may optionally choose to warm-up its *StatsTable* by exchanging data with its network bootstrap.

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 perform the best for this request.

Formally, given an array of statistical data d_i (e.g. a DHT's query latency history), we define M to be a set of metric functions $(m_1, m_2, \ldots, m_k \in M)$, where $m_i(d_i, \Delta t) \rightarrow \mathcal{R}$ (Δt is an optional time interval). The cost function is then weighted by:

$$C_i(M) = \alpha_1 \cdot m_1 + \alpha_2 \cdot m_2 + \ldots + \alpha_k \cdot m_k$$

Where

$$\sum_{i=1}^{k} \alpha_i = 1$$

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 $R(\mathcal{C}, C_i) \rightarrow Bool$.

$$\bigcup_{i=1}^{j} R(\mathcal{C}, C_i)$$

provides a set of booleans that determines which DHT(s) the query should be routed on. Note that this decision can range from only one substrate to all DHTs. 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.

Type	Metrics
Media	Size, Latency, Success
DNS	Latency, Success
Cache	Freshness, Latency, Sucess
Documents	Popularity, Latency, Success

Table 1. Example Metrics

4. 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. This section provides a description of the details of this prototype implementation.

4.1. Simulator

All experiments use a discrete-event, link-layer simulator for evaluation. The simulator emulates a physical node-network and provides a simple interface for multiple overlays to exist on a single node. These overlays are unaware of each other because incoming messages are dispatched to the correct overlay by using a system of message domains. Every node is assigns a unique network address specified

by a *NodeID*. To simplify the implementation both node's ID and object's ID use the same construction.

The simulator is run by a driver that also prepares the events and builds the network topology. Currently, the driver parses *GT-ITM* [1] generated topologies and custom-written event files. Due to memory constraints, the simulator is currently limited to topologies of only a few thousands nodes.

4.2. Pastry

The Pastry implementation was adopted from the open-source FreePastry[2] project written at Rice University. Like the actual implementation, the Pastry overlay keeps a routing table, a leaf set, and a neighborhood set. When inserting a filetuple, the Pastry implementation will wrap the insert with a routing message and forward it to a node with an ID at least one digit more similar to the key than itself. By routing messages in this manner, Pastry can guarantee message will arrive at its destination in $\lceil log_{2^b} N \rceil$ hops, where b is a network parameter. Throughout our simulations, we set $b = 4$. The key will eventually be inserted into the node with the *NodeID* closest the filetuple's key. Lookups are performed in a similar manner. Under failure, the Pastry implementation might try to reroute the message using an alternate path or hand it off to a member of its leaf set.

Using the FreePastry implementation as a guideline, stale routing table and leaf set entries are removed by checking the availability of the nodes via pinging. Furthermore, a maintenance procedure that exchanges entry information with neighbors is periodically run to update the tables.

In spite of this, Pastry tends to be a more reactive DHT and performs most of its work upon node entries and exits. This reduces the message overhead, but increases the average latency of a query. In addition, a higher hop count will decrease the reliability of a query under high levels of churn. This is due to the increased probability that an intermediate node will be unavailable.

4.3. Kelips

The Kelips overlay was derived from the original C WinAPI implementation and the corresponding technical paper [9]. Similar to the original implementation, Kelips revolves around a $\sqrt{N}$ affinity groups and active gossiping. Each member belongs to an affinity group by hashing its *NodeID* between $(0 - \sqrt{N})$ and maintains information about other group members. In addition, three contacts are kept from other affinity groups. When a filetuple is inserted or queried, the key will be hashed using the same manner as the *NodeID* and forwarded to a contact known in that group. When faced with failure, our implementation allows retries to be passed around at both the source's affinity group and the destination's group for a specific TTL. This helps overcome problems with empty contact lists.

Kelips uses an active approach to maintaining node and filetuple entries. Every 2 seconds, a node will gossip some information to other members in its affinity group and a few contacts. These messages are considered "heartbeats" to the receiver. If a heartbeat is not heard after a specified timeout period, Kelips will consider the entry to be stale and remove it from the table. Refer to the paper for specific detail about the gossiping protocol.

The major strength of Kelips is the $O(1)$ lookup costs and fault-tolerance obtained from gossip protocols. However, in addition to moderate message overhead, newly inserted objects are typically unavailable until the update has propagated throughout the affinity group. When faced with high level churn, this can become even more apparent due to the inconsistent affinity group view.

4.4. MultiRouter

Our MultiRouter prototype was implemented as an overlay very similar to Kelips and Pastry, but it interacts with only its DHT substrates and not with the underlay physical network. A *join* request on the MultiRouter prototype will initiate a join on both Kelips and Pastry and reset the filetuple metrics data. Although, it might be beneficial to warm up the metrics using neighbors' data, we decided to leave this aspect for future research. Once all the substrates indicate they are ready for operation, the MultiRouter will accept requests and inserts from the application. If a MultiRouter happens to leave the network, it will keep all of the old Kelips and Pastry routing and filetuple information that was present upon departure. In the prototype, this data is reused immediately when the MultiRouter rejoins at a later time.

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. Since latency is considered to be more variable than the success rate, recent failures have a greater weight than fluctuations in latency. Adversely, old failures should have little impact on the MultiRouter's decision. To smooth out the fluctuations of varying latency, our prototype employs an aging function. Given a new latency datum $l \in [0, 1]$

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

[2] http://freepastry.rice.edu

(where 1 is proportional to a *timeout*) and $\beta \in [0, 1]$:

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

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. Note that by using an aging algorithm, we can compact all the latency data into a single value $latency_i$ instead of using an array.

For the failure rate, our prototype uses a decaying function dependent on the current time, t. This enables the characteristic described above that only recent failures are relevant to the decision-making process. Given a new failure rate datum $f \in \{0, 1\}$, where $f = 1$ upon a returned failure or *timeout* and 0 otherwise and $\gamma \in [0, 1]$:

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

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 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.

Given these two metrics we can compute the cost function for a given DHT. Derived from the equation in Section 3, the cost function for our implementation is

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

where $m_1 = latency_i$ and $m_2 = failure_i$.

Since both metrics are equally important, we set $\alpha_1 = \alpha_2 = 0.5$ in our prototype. This give us the relative "cost" of using this prototype. Obviously, a lower cost is always preferred, but there are many cases when you wish to include all the DHTs regardless of the reported cost.

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.

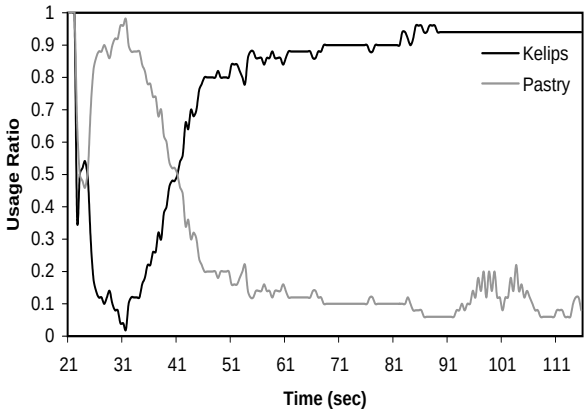
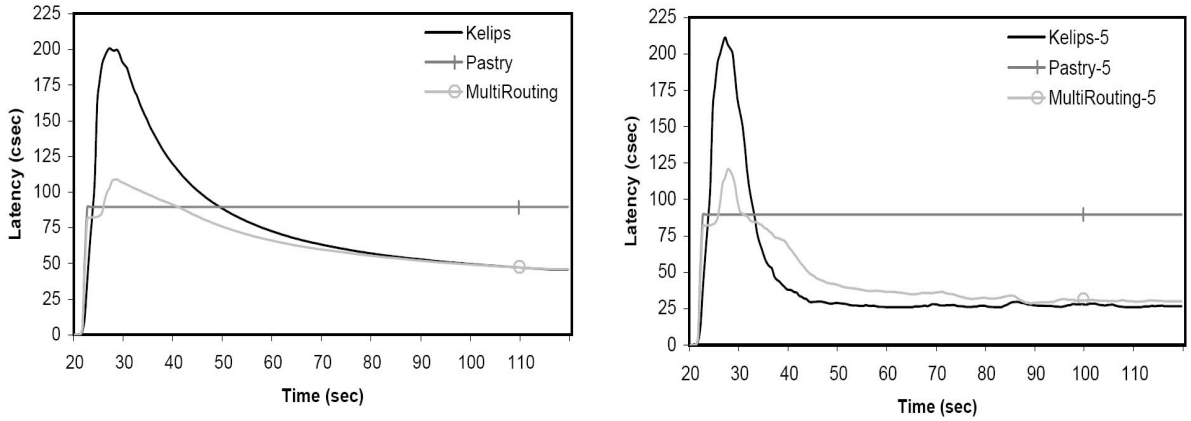


Figure 2. MultiRouter Adaptivity: Right after an object is inserted, the MultiRouter prefers the Pastry for lookups, since it is faster. However, after the object meta-data has been replicated enough for the Kelips substrate to have better lookup times, the MultiRouter switches over.

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 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.

5. Experimental Results

Using the simulator and prototype described in Section 3, we evaluated a MultiRouter implementation using simulation runs and trace-based experiments. We analyze three different aspects of the MultiRouter design. First, a simple micro-benchmark is run to show the basic properties of its behavior under ideal conditions. We then evaluate the performance of the MultiRouter under levels of generic moderate churn. To show more realistic conditions, we also present a trace-based experiment with a larger topology. Finally, we analyze the potential increase in overhead that can occur from using the MultiRouting approach. In the last experiment we use the metric "messages per second" because the average bandwidth per message is approximately



(a) Average Latency

(b) 5-Avg Latency

Figure 3. Average and 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. Note that the marks on the remaining graphs are for differentiation purposes only.

equivalent (50 - 75 B/msg) across all implementations. The experiments were run on a 3.0 GHz WinXP P4 with 512 MB of RAM.

While some of the simulator parameters vary between experiments, the Kelips and Pastry network parameters remained constant throughout the evaluation. We attempted to use what are considered normal parameters indicated by their respective papers. For our Kelips implementation, we always pre-calculated the value of $k = \sqrt{N}$. In addition, we set the gossiping interval to be every 2 seconds, restricted the number of contacts per gossip message to 3, and the number of entries to 10 per message. Also, each node was bounded to a maximum of 3 contacts per foreign affinity group. This gives a typical message size of 13 entries. For Pastry, we set the value of $b = 4$ and $L = 16$, which also creates a typical message size of 13 entries.

Our results show that the MultiRouter is capable of improving both the latency and success rate of DHTs under churn. However, it should be clear that the MultiRouter's success relies on its adaptive properties that are ultimately dependent on the performance of the underlying DHTs. Therefore, the MultiRouter will never outperform all its DHTs substrates at any given instance, but instead generally provide the "best of all worlds" in performance and availability. Also note that in the following experiments we insert all of the keys prior to the first query that is executed. While this approach is not as realistic, it is still representative of the behavior that would occur on a per-key basis even if the keys were to be inserted continuously throughout the experiment.

5.1. Micro-Benchmark

The micro-benchmark is used to present simple MultiRouting behavior as a "proof-of-concept." The micro-benchmark uses a trans-stub topology of 200 nodes using the *GT-ITM* generator. Each node in the network ran an instance of the MultiRouter and the two DHTs, Kelips and Pastry. 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. While this is a bit excessive because most applications would cache multiple lookups, it could be relevant to an application like cooperative-caching where freshness of content is important and certain content is fetched at high-rates due to its Zipf-like distribution. Regardless, the purpose of this experiment is to show a basic adaptability case that the MultiRouter can handle.

Since the query stream is initiated immediately after the file has been inserted, the Kelips overlay lacks the time needed to disseminate this information throughout the affinity group. On the other hand, Pastry is perfectly capable of retrieving the filetuple immediately. Figure 2 shows the DHT usage ratio by the MultiRouter. This graph indicates behavior that is expected using this reasoning. 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.

Note that the ratios do not always add up to 100 percent because more than one overlay can be selected for a given query. This is apparent at the beginning of the query stream

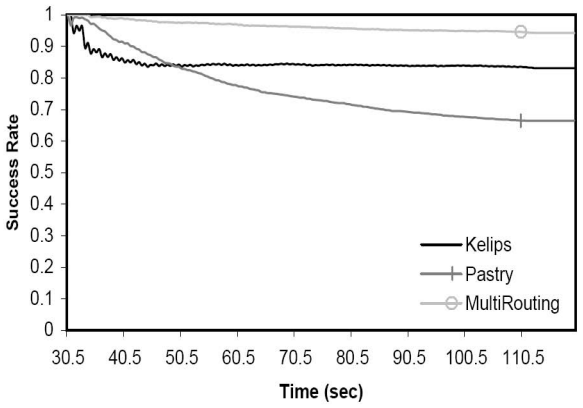


Figure 4. Effect of Generic Churn: The MultiRouter is able to guarantee lookup success rates better than either Kelips or Pastry when there is churn. This is due to replicated files and adaptive lookups.

where the MultiRouter is warming-up its metrics. Also note the blip in the Pastry plot near the end of the run. This is due to the coldness rule described in the previous section. The MultiRouter is testing to see if Pastry's performance has changed since it was last used.

We also evaluated the performance of the MultiRouter compared to the individual DHTs. We ran each overlay on an average of 50 times using the same scenario and computed the average latencies of the lookups. The results are shown in Figure 3(a). It is evident from this plot that the MultiRouter generally has the lowest average latency per lookup. However, due to the dynamic behavior of the latencies, we also recalculated the latency averages to only include the last 5 queries. This gives a better picture to how well an overlay is performing at a given point in time. Figure 3(b) presents these results, which are very similar to Figure 3(a). However, note that now Kelips does slightly better than the MultiRouter after the large spike. This is the interval where the Kelips cost function is getting progressively better until it finally exceeds Pastry. Also note that the switch doesn't happen instantaneously because the MultiRouter needs to wait for the Kelip's failure costs to decay back to zero. In the latter part of the graph, the MultiRouter's performance converges to the Kelips latency.

5.2. 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 point of this simulation is to see how much performance improvement we can get under stressful network conditions. Again we compared this to a standard Kelips and Pastry implementation.

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. This type of situation favors Kelips because of its proactive gossiping architecture provides additional reliability to failures and churn.

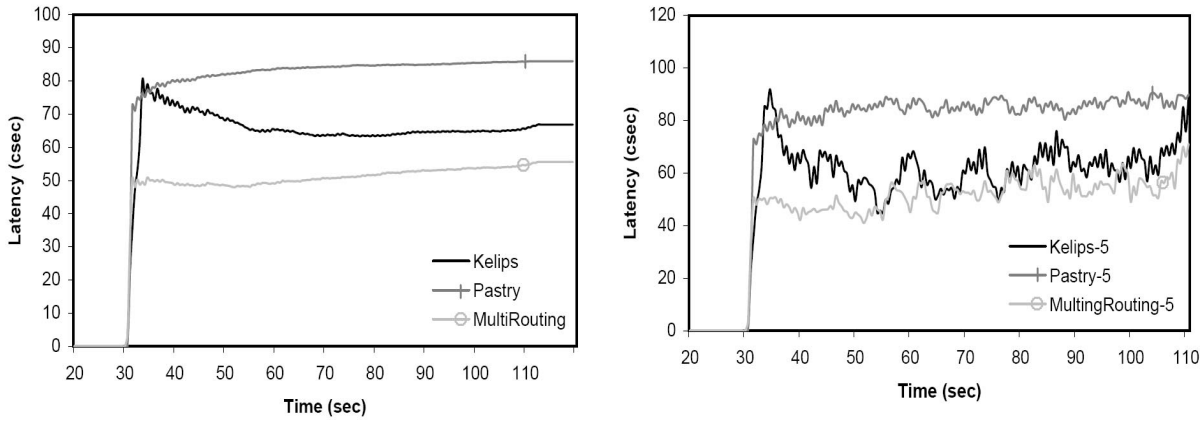
However, it is plausible that the MultiRouter achieves a better success rate by retrying a failed query many times on alternate overlays. As a result, the latency of the lookup could be adversely affected by an increased success rate. To determine if this is the case, we also calculated the average latency of the lookups. Similar to the micro-benchmark, we provide two graphs: the total average latency (Figure 5(a)) and the average latency of the last 5 queries (Figure 5(b)). Fortunately, the MultiRouter performs better of the three in both analysis. It is evident from the figure that the MultiRouter follows the same line structure as Kelips. We hypothesize that this is because the MultiRouter tends to favor the Kelips substrate during most runs, but achieves the additional performance boost by using Pastry during periods of poor Kelips performance.

5.3. Overnet Trace

To simulate a more realistic churn scenario, we also ran a similar experiment using availability traces from the Overnet P2P network [4]. The actual trace pinged approximately 3000 nodes per hour on the Overnet network to check their status. We mapped 500 of these traces to a 1275 trans-stub topology. The remainder of the nodes remained alive during the entire experiment. We also inserted 100 filetuples onto the always-alive nodes of the network. This guarantees that at least 1 node with the filetuple is alive in the network. Similar to the previous experiment we queried two filetuples every 0.5 sec from random nodes.

To align this trace to the previous experiment we scaled-down the interval of the trace to a per second basis. We ran the trace for about 5 simulated minutes and got the success rates shown in Figure 6(a). Again, we see that the MultiRouter increases the probability of success by an average of 10 percent compared to Kelips and 20 percent better than Pastry.

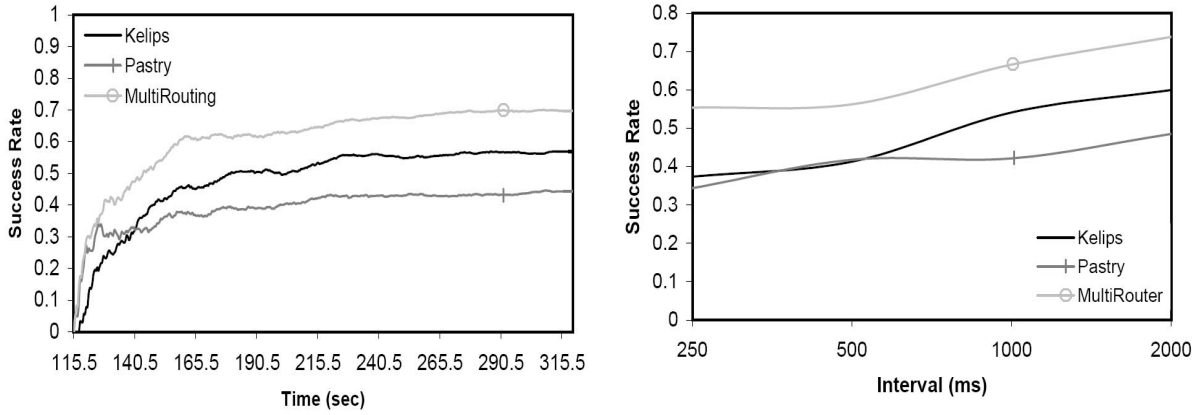
Since the choice of a 1 second interval for the trace data is arbitrary, we reran this experiment and varied the trace interval between .25 seconds to 2 seconds and calculated the average success rate of each overlay. These results are presented in Figure 6(b). In general the same relationship



(a) Average Latency

(b) 5-Avg Latency

Figure 5. Average and 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.



(a) 1-second

(b) Varying Interval

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

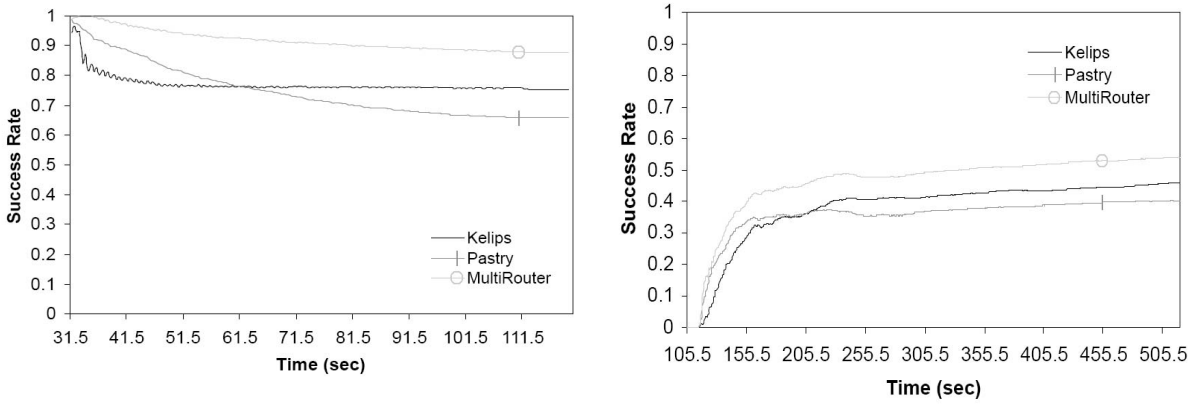
holds between Kelips, Pastry, and the MultiRouter. We also see that the increase in churn adversely affect the success of all implementations. This behavior is expected.

5.4. Message Overhead

The caveats of the MultiRouter are the increase in resource usage at individual nodes, i.e. bandwidth and memory utilization. For example, in the previous experiments the MultiRouter was using the two DHT substrates, Kelips and Pastry, as is. If the overhead of using Kelips is (3.90 msg/sec) and Pastry is (0.46 msg/sec), then the overhead of the MultiRouter is simply the summation of the component's requirements (4.36 msg/sec).

One might consider this an unfair assessment since the amount of bandwidth the three systems are using is uneven. Therefore, we tweaked each system to produce approximately the same amount of messages per second (3.6 msg/sec) and reran the experiments. To accomplish this we had to increase the gossiping interval on Kelips to 2.75 seconds and increase the number of maintenance messages in Pastry to occur every 4 seconds. Since the MultiRouter has to account for both Kelips and Pastry messages, we had to increase the gossiping interval even further to 3.5 seconds and increase the Pastry maintenance interval to occur once every couple of minutes.

As expected, we saw similar behavior patterns as before but the differences in performance are somewhat less.



(a) Generic Churn Success Rate

(b) Overnet Trace Success Rate (1 sec interval)

Figure 7. Success Rate with Equal Number of Messages: The same pattern is displayed from the previous experiments, but the difference in rates is reduced. This is to be expected.

For example, Figure 7(a) and Figure 7(b) display the success rate for the generic churn and Overnet 1-second experiments discussed in the previous sections. The MultiRouter still performs the best of the three but the success rates of the MultiRouter and Kelips has been reduced slightly. Another example includes the average latency and the average latency of the last 5 queries from the generic churn experiment in Figure 8(a) and Figure 8(b). The MultiRouter still does a little bit better on average, but the difference if any is harder to discern especially when using the average of the last 5 queries.

Obviously modifying the DHT parameters in this fashion will adversely affect the MultiRouter's performance because ultimately the MultiRouter relies on the underlying substrate to perform the actual operation on its behalf. However, it is evident that even in this extremely contrived example with more poorly functioning DHTs, it is still able to maintain an advantage over the straight DHT implementations with better performing parameters.

6. 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 with little additional complexity. 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.

While we were able to evaluate the performance of a specific implementation of the MultiRouter using our prototype described in the paper, additional work needs to be done to study in more detail the parameter space of different metric and DHT combinations and weights. This will likely improve performance gains seen in this paper and show how more realistic applications would perform using MultiRouting. In addition, this work may lead to the development of an expressive language that would allow communication between the user application and the MultiRouter. With such a communication mechanism, dynamic metrics and rule sets could be employed using feedback from the MultiRouter further improving its adaptive qualities.

References

[1] A. Akella, B. Maggs, A. Seshan, A. Shaikh, and R. Sitaraman. A Measurement-Based Analysis of Multihoming. *Proceedings of ACM SIGCOMM*, Aug 2003.

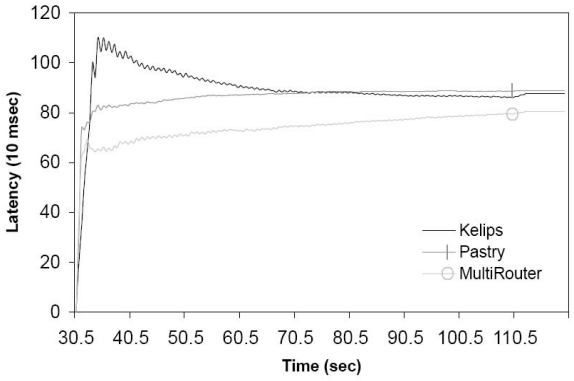
[2] A. Akella, J. Pang, J. Pang, S. Seshan, and A. Shaikh. A Comparison of Overlay Routing and Multihoming Route Control. *Proceedings of ACM SIGCOMM*, Aug 2004.

[3] D. Anderson, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. *Proceedings of SOSP '01*, Oct 2001.

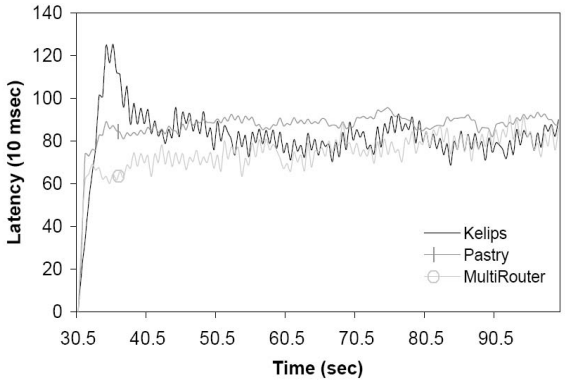
[4] R. Bhagwan, S. Savage, and G. Voelker. Understanding Availability. *Proceedings of International Workshop on Peer-to-Peer Systems*, Feb 2002.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks. *Proceedings of 10th European SIGOPS Workshop*, Sept 2002.

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



(a) Generic Churn Average Latency



(b) Generic Churn 5-Average Latency

Figure 8. Average Latency with Equal Number of Messages: The differences in latencies between the three implementations become less noticeable.

[7] D. Goldenberg, L. Qiu, H. Xie, Y. Yang, and Y. Zhang. Optimizing Cost and Performance for Multihoming. *Proceedings of ACM SIGCOMM '04*, Aug 2004.

[8] A. Gupta and B. Liskov. One Hop Lookups for Peer-to-Peer Overlays. *Proceedings of Hot Topics in Operating Systems*, 2003.

[9] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead. *Proceedings of International Workshop on Peer-to-Peer Systems*, Feb 2003.

[10] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. *Proceedings of International Workshop on Peer-to-Peer Systems*, Feb 2004.

[11] P. Linga, I. Gupta, and K. Birman. A Churn-Resistant Peer-to-Peer Web Caching System. *Proceedings of ACM Work-Shop SSRS*, Oct 2003.

[12] B. Loo, J. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. *Proceedings of the 30th VLDB Conference*, 2004.

[13] S. Marti, P. Ganesan, and H. Garcia-Molina. DHT Routing Using Social Links. *Proceedings of International Workshop on Peer-to-Peer Systems*, Feb 2004.

[14] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. *Proceedings of ACM SIGCOMM*, Aug 2003.

[15] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. *Proceedings of the USENIX Annual Technical Conference*, June 2004.

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

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