# Mercury: Supporting Scalable Multi-Attribute Range Queries

Ashwin R. Bharambe Mukesh Agrawal
Srinivasan Seshan

January 2004
CMU-CS-□□-□□□□

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

This paper presents the design of Mercury, a scalable protocol for supporting multi-attribute range-based searches. Mercury differs from previous range-based query systems in that it supports *multiple attributes* as well as performs explicit load balancing. Efficient routing and load balancing are implemented using novel light-weight sampling mechanisms for uniformly sampling random nodes in a highly dynamic overlay network. Our evaluation shows that Mercury is able to achieve its goals of logarithmic-hop routing and near-uniform load balancing.

We also show that a publish-subscribe system based on the Mercury protocol can be used to construct a distributed object repository providing efficient and scalable object lookups and updates. By providing applications a range-based query language to express their subscriptions to object updates, Mercury considerably simplifies distributed state management. Our experience with the design and implementation of a simple distributed multiplayer game built on top of this object management framework shows that indicates that this indeed is a useful building block for distributed applications.

**Keywords:** Range queries, Peer-to-peer systems, Distributed applications, Multiplayer games

# 1 Introduction

Much recent work on building scalable peer-to-peer (P2P) networks has concentrated on Distributed Hash Tables or DHTs [23,24,26]. DHTs offer a number of scalability advantages over previous P2P systems (e.g., Napster, Gnutella, etc.) including load balancing and logarithmic hop routing with small local state. However, the hash table or "exact match" interface offered by DHTs, although fruitfully used by many systems [5–7], is not flexible enough for many applications. For example, it is unclear how DHTs could be applied to regain the highly desirable flexibility offered by keyword-based lookups of file-sharing applications.

The main contribution of this paper is the design and evaluation of Mercury, a scalable routing protocol, for supporting *multi-attribute range queries*. In our model, each query is a *conjunction of ranges* in one or more attributes. The attributes not present in the query are assumed to be wildcards. We believe range queries significantly enhance search flexibility in a number of scenarios. In addition to being useful for answering user queries, we find that range-based queries can also be useful in the construction of many distributed applications. Using a publish-subscribe system based on Mercury, we show that an application can flexibly and scalably manage updates and queries to distributed application state. We also present the design and testbed evaluation of a distributed multiplayer game built using the above framework.

A number of recent systems [11, 12, 16] have proposed distributed protocols which support range-based queries. Mercury mainly differs from these systems in that it supports *multi-attribute* range-based queries *and* explicit load balancing.

There are two main components of Mercury's design. First, Mercury handles multi-attribute queries by creating a *routing hub* for each attribute in the application schema. Each routing hub is a logical collection of nodes in the system. Queries are passed to exactly one of the hubs corresponding to the attributes that are queried, while a new data item is sent to all hubs for which it has an associated attribute. This ensures that queries retrieve all relevant data items present in the system.

Second, for supporting range queries, Mercury organizes each routing hub into a circular overlay of nodes and places data *contiguously* on this ring, i.e., each node is responsible for a *range* of values for the particular attribute. While the notion of a circular overlay is similar in spirit to most existing DHT designs, Mercury cannot use randomizing hash functions for placing data. For supporting range queries, Mercury *requires* that data be placed contiguously. This requirement introduces a fundamental challenge: because Mercury cannot use hash functions, data partitioning among nodes can become non-uniform (as we explain in Section 3.2) requiring an explicit load-balancing mechanism. However, the load-balancing mechanism is fundamentally incompatible with many of the techniques that DHTs use to guarantee routing efficiency.

The solution of the above challenges forms the core contribution of this paper. Some of the interesting algorithms incorporated in Mercury include:

- A message routing algorithm that supports range-based lookups within each routing hub in $\mathcal{O}(\log^2 n/k)$ when each node maintains $k$ links to other nodes.
- A low-overhead random sampling algorithm that allows each node to create an estimate of system-wide metrics such as data value and load distribution.
- A load-balancing algorithm (which exploits the random sampling algorithm) that ensures that routing load is uniformly distributed across all participating nodes.

- An algorithm for reducing query flooding by estimating how selective each of the predicates in a query are based on past database insertions.

In addition to describing the design of Mercury, we also explore how the added flexibility provided by range query lookups can simplify the construction of distributed applications, especially the state management tasks of such applications. We observe that an instance of a distributed application, at any single point of time, is typically interested [1] in a small subset of the entire application state. Moreover, this subset is typically not random – the objects are related in some manner. For example, in each instance of a distributed multiplayer game, a player is only interested in the virtual entities in her room or arena. We show that, using a range query based publish-subscribe [2, 3] system built on top of Mercury, we can provide efficient and scalable management of distributed object updates and lookups. We believe that a *range query* significantly enhances the application's ability to accurately express its interests.

As a proof-of-concept system, we have implemented a simple Asteroids-like multi-player distributed game using the publish-subscribe object repository framework outlined above. Each player in the game "subscribes" to the region of the game near her ship. As a result, messages that are needed to update a particular player's screen are automatically delivered to the host. Our testbed evaluation indicates that Mercury reduces bandwidth usage by implementing effective filtering and provides low routing-hop delivery for such applications.

The remainder of the paper is organized as follows. In the next section, we compare Mercury to prior related work in this area. Section 3 details the basic Mercury protocol for routing data-records and queries. Section 4 presents enhancements which improve the performance of the basic protocol. In Section 5, we evaluate the scalability and load-balancing properties of the Mercury system. In Section 6, we present the design of our publish-subscribe based object repository framework and proof-of-concept distributed game. Finally, Section 7 concludes.

## 2 Related Work

In this section, we compare and contrast our approach to implementing range queries with that of related systems. Our discussion focuses on two fundamental questions:

- ▷ Can we use *existing DHTs as building blocks* to implement range query predicates?
- ▷ How is our design different from other systems like SkipNet [11], etc., which also provide range query support?

### Using existing DHTs for range queries

A large number of distributed hash table designs [11, 23, 24, 26] have been proposed over the past few years. They provide a hash table interface to the application, viz., `insert(key, value)` and `lookup(key)` primitives. Recent research [4, 10] has shown that, in addition to the basic scalable routing mechanism, DHTs offer much promise in terms of load balancing, proximity-based routing, static resilience, etc. Hence, it is a natural question to ask if we could implement range queries using just the `insert` and `lookup` abstractions provided by DHTs.

Our analysis, based on analyzing possible strawman designs using DHTs, indicates that the abstractions provided by a DHT are not enough for implementing range queries. Fundamental

[1] We say that an application instance is 'interested' in an object, if it wishes to keep it in an updated state.

to our argument is the fact that all existing DHT designs use randomizing hash functions for inserting and looking up keys in the hash table. While hashing is crucial for DHTs in order to get good load balancing properties, it is also the main barrier in using a DHT for implementing range queries. This is because the hash of a range is not correlated to the hash of the values within a range. Hence, it is necessary to create some artificial correlation between ranges and values which is invariant under hashing.

One natural way to achieve this is to *partition the value space into buckets* and map values and ranges to one or more buckets. A bucket forms the lookup *key* for the hash table. Then, a range query can be satisfied by simply performing lookups on the corresponding bucket(s) using the underlying DHT. However, this scheme has several fundamental drawbacks. It requires the application to *a priori* perform the partitioning of space. This can be very difficult or even impossible for many applications, e.g., partitioning of file names. Moreover, load balancing and query performance is highly dependent on the way partitioning is performed. For example, if the number of buckets is too small, i.e., the partition is too coarse, queries will get mapped to a smaller set of nodes creating load imbalance. Increasing the number of buckets, on the other hand, will increase the routing hops required to answer a range query.

This indicates that while a DHT-based scheme may not be an impossibility, its implementation is likely to be *awkward* and complicated. By avoiding randomizing hash functions, Mercury seeks to remove this difficulty. At the same time, we also note that the design of Mercury is inspired from and similar in many respects to existing DHT designs. Hence, we believe that it can easily build upon recent advances in proximity-based routing and achieving resilience in DHTs [10].

**Comparison with systems supporting range queries**

In this section, we compare Mercury against recent systems which offer range query support. These include SkipNet [11], PIER [12] and DIM [16].

The SkipNet DHT organizes peers and data objects according to their lexicographic addresses in the form of a variant of a probabilistic skip list. It supports logarithmic time range-based lookups and guarantees path locality. Mercury is more general than SkipNet since it supports range-based lookups on *multiple-attributes*. Our use of random sampling to estimate query selectivity constitutes a novel contribution towards implementing scalable multi-dimensional range queries. Load balancing is another important way in which Mercury differs from SkipNet. While SkipNet incorporates a constrained load-balancing mechanism, it is only useful when part of a data name is hashed, in which case the part is inaccessible for performing a range query. This implies that SkipNet supports load-balancing *or* range queries – not both.

One might argue that the query-load imbalance in SkipNet can be corrected by using *virtual servers* as suggested in [22]. However, it is unlikely to help in this regard for the following reason: for effective load-balancing, the number of virtual servers needed must be proportional to the *skew* (ratio of *max* to *min*) in the load. The scheme proposed in [22] assumes that the load skew results from the standard deviation of random hash function distributions, which is typically very small($\mathcal{O}(\log n)$, see [1]). However, in our case, the skew results from differences in *query workload*, which can grow quite large. Hence, larger number of virtual servers would be required increasing routing hops by about $\log(s)$ where $s$ is the skew. Moreover, the scheme would fare even worse for range lookups since it would increase the number of distinct nodes accessed for processing the query by $\mathcal{O}(s)$.

The PIER system is a distributed query engine based on DHTs. However, for all queries except

exact match lookups, it uses a *multicast* primitive which performs a controlled flood of the query to all nodes within a particular namespace.

The DIM data structure [16] supports routing multi-dimensional range queries by embedding them into a two-dimensional space and using a geographic routing algorithm. However, the routing cost scales only as $\mathcal{O}(\sqrt{n})$, which while reasonable in a medium-sized sensor network, is quite expensive for larger scales. Furthermore, the "volume expansion" that occurs while projecting from higher dimensions onto two-dimensions can be quite large resulting in more flooding of the query. Also, queries containing wildcards in certain attributes get flooded to all nodes. On the other hand, Mercury, like most databases, uses query selectivity mechanisms to route through the attribute hubs of highest selectivity thereby significantly reducing flooding for queries containing only a few attributes.

All the above systems and recent work on balancing load in such systems [1, 22] treat load on a node as being proportional to the *range of identifier values* the node is responsible for. In other words, they assume a uniform data distribution which is sensible for DHTs which use cryptographic hashes. Mercury, on the other hand, defines load on a node as the number of messages routed or matched per unit time, and supports explicit and flexible load balancing. We note that Mercury uses a *leave-join* style load balancing algorithm that is similar to [22]. Karger and Ruhl [13] have independently discovered a similar join-leave based load balancing mechanism. However, their protocol requires communication with $\log n$ *random* nodes in the system. In the face of skewed node range distributions, sampling nodes uniformly at random is far from trivial. A significant part of the Mercury protocol is aimed at addressing this difficulty. In general, many approaches to diffusion-based dynamic load balancing [9] use hard-to-get information about a dynamic distributed network to make an informed decision. Instead, Mercury uses light-weight sampling mechanisms to track load distribution within the overlay.

# 3 Mercury Routing

In this section, we provide an overview of the basic Mercury routing protocol. We note that this basic routing only works well in a limited set of conditions. Later, in Section 4, we significantly extend the capabilities of this basic routing to work for a wider set of operating points.

## 3.1 Data Model

In Mercury, a data item is represented as a list of typed attribute-value pairs, very similar to a record in a relational database. Each field is a tuple of the form: (`type`, `attribute`, `value`). The following types are recognized: `int`, `char`, `float` and `string`.[2]

A query is a *conjunction* of predicates which are tuples of the form: (`type`, `attribute`, `operator`, `value`). A disjunction is implemented by multiple distinct queries. Mercury supports the following operators: $<, >, \leq, \geq$ and $=$. For the `string` type, Mercury also permits *prefix and postfix* operators. Figure 1 presents an example.

[2]Our basic data types are sortable, enabling us to define numeric operations (addition and subtraction) on them. Care needs to be taken when handling `string` attributes.

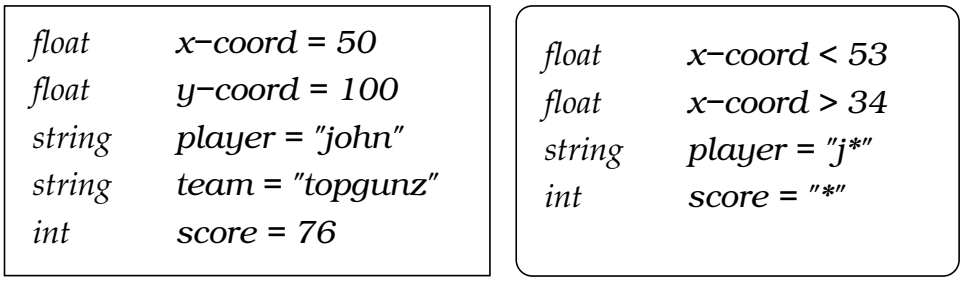



**Figure 1: Example of a data item and a query as represented in the Mercury system.**

### 3.2 Routing Overview

Mercury supports queries over multiple attributes by partitioning the nodes in the system into groups called *attribute hubs*. This partition is only logical, i.e., a physical node can be part of multiple logical hubs. Each of the attribute hubs is responsible for a specific attribute in the overall schema. Hubs can be thought of as orthogonal dimensions of a multi-dimensional attribute space. The first routing hop determines which dimension to route through. The rest of the routing is unidimensional and is based on the values of a single attribute of the data item. We note that this mechanism does not scale very well as the number of attributes in the schema increase; however, we believe that the schemas of many applications are likely to be reasonably small.

To simplify the description, we will use the following notation: let $\mathcal{A}$ denote the set of attributes in the overall schema of the application. $\mathcal{A}_Q$ denotes the set of attributes in a query $Q$. Similarly, the set of attributes present in a data-record $D$ is denoted by $\mathcal{A}_D$. We use the functions $\pi_a$ to denote the value (range) of a particular attribute $a$ in a data-record (query).We will denote the attribute hub for an attribute $a$ by $H_a$.

Nodes within a hub $H_a$ are arranged into a circular overlay with each node responsible for a *contiguous range* $r_a$ of attribute values. A node responsible for the range $r_a$ *resolves* all queries $Q$ for which $\pi_a(Q) \cap r_a \neq \phi(= \{\})$, and it *stores* all data-records $D$ for which $\pi_a(D) \in r_a$. Ranges are assigned to nodes during the join process.

#### Routing Queries and Data-Records

Queries are passed to exactly one of the hubs corresponding to the attributes that are queried. Specifically, a query $Q$ is delivered to $H_a$, where $a$ is *any* attribute chosen from $\mathcal{A}_Q$. We will see in Sections 4.3 and 5.4 that although choosing any attribute hub suffices for matching correctness, substantial savings in network bandwidth can be achieved if the choice is done more intelligently using query selectivity. Within the chosen hub, the query is delivered and processed at all nodes that could potentially have matching values.

To guarantee that queries locate all the relevant data-records, a data-record $D$, while insertion, is sent to *all* $H_b$ where $b \in \mathcal{A}_D$. This is necessary because the set of queries which could match $D$ can arrive in any of these attribute hubs. Within each hub, the data-record is routed to the node responsible for the record's value for the hub's attribute.

Notice also that we could have ensured correctness by sending a data-record to a single hub in $\mathcal{A}_D$ and queries to all hubs in $\mathcal{A}_Q$. At first glance, this might appear to be a better choice

since data-records could be much bigger in size than queries and replicating them might be more expensive. However, recall that a query can get flooded to multiple locations within each hub depending on its selectivity. This, combined with the fact that many queries may be extremely non-selective in some attribute (thereby, flooding a particular hub), led us to choose a design with data-records broadcast to all hubs. Mercury could be easily modified to support a situation where this problem does not exist.

Within a hub $H_a$, routing is done as follows: for routing a data-record $D$, we route to the *value* $\pi_a(D)$. For a query $Q$, $\pi_a(Q)$ is a range. Hence, for routing queries, we route to the *first* value appearing in the range and then use the contiguity of range values to spread the query along the circle, as needed.

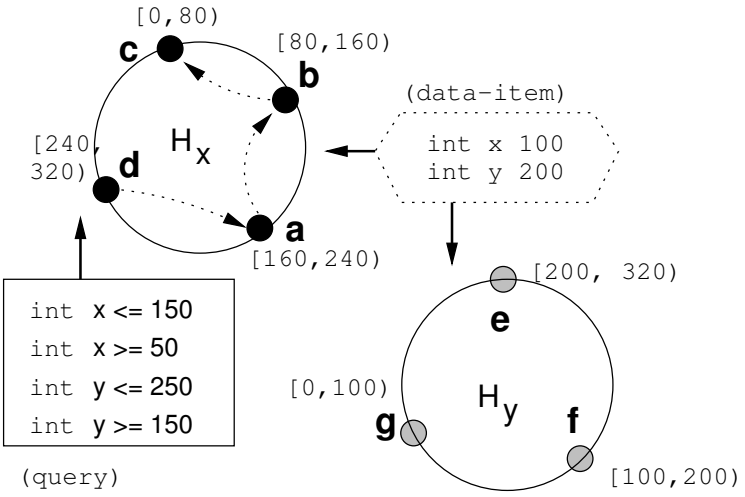



**Figure 2: Routing of data-records and queries.**

Fig 2 illustrates the routing of queries and data-records. It depicts two hubs $H_x$ and $H_y$ which may correspond to, for example, X and Y coordinates of objects. The minimum and maximum values for the $x$ and $y$ attributes are 0 and 320 respectively. Accordingly, the ranges are distributed to various nodes. The data-record is sent to both $H_x$ and $H_y$, where it is stored at nodes `b` and `e`, respectively. The query enters $H_x$ at node `d` and is routed (and processed) at nodes `b` and `c`.

This routing places one additional requirement on the connectivity of each node. In addition to having a link to the predecessor and successor within its own hub, each node must also maintain a link to each of the other hubs. We expect the number of hubs for a particular system to remain low, and, therefore, do not expect this to be a significant burden. We discuss the maintenance of these links later in the section.

### Design Rationale

In this section, we discuss some of the promising alternative designs for implementing a distributed multi-attribute range-based search and comment qualitatively on the trade-offs involved.

Many DHTs [18, 26] use a cryptographic hash or random value to give IDs to nodes and data stored in the DHT. However, Mercury does not use any such cryptographic hashes or random values. This simpler mapping of data and nodes in the system allows the lookup of range predicates in subscriptions to a collection of contiguous nodes in a hub. We note that one of the main

purposes of using a cryptographic hash in existing DHTs is to assign data to nodes uniformly and randomly[3]. The elimination of this randomness makes load-balancing in Mercury a concern. Since there are likely to be particular ranges of an attribute that are more popular for publications and subscriptions, nodes responsible for these ranges from will be unfairly overloaded with both routing and computation tasks. Mercury performs explicit load balancing (see Section 4.4) by moving around nodes and changing their responsibilities according to the loads. This enables the combination of good load-balancing with support for range predicates However, one important side effect is that the distribution of range sizes is no longer guaranteed to be uniform.

With the removal of cryptographic hashes, we could have used a variety of different DHTs as the basis for our design. Our design treats the different attributes in an application schema *independently*, i.e., routing a data item $D$ within a hub for attribute $a$ is accomplished using only $\pi_a(D)$. An alternate design would be to route using the values of all attributes present in $D$, e.g., treating each attribute as a CAN dimension [23]. Since each node in such a design is responsible for a value-range of *every* attribute, a subscription that contains a *wild-card* attribute can get flooded to all nodes. We could have merged dimensions like in the DIM data structure [16] but this would still have had similar problems for queries covering large areas. By making the attributes independent, we restrict such flooding to at most one attribute hub. Furthermore, it is quite likely that some other attribute of the query is more selective and by routing the query to that hub, we can eliminate flooding altogether.

### 3.3 Constructing Efficient Routes

Recall that most of the routing in Mercury occurs within an attribute hub (only the first hop crosses hubs.) Thus, it is essential that the overlay structure for each attribute hub be scalable and efficient.

Simply using successor or predecessor pointers can result in $\theta(n)$ routing delays for routing data-records and queries. Like Symphony [18], the key to Mercury's route optimization is the selection of $k$ *long-distance* links that are maintained in addition to the successor and predecessor links. As a result, each node has a routing table of size $k+2$ including its neighbors along the circle. $k$ is a configurable parameter here and could be different for different nodes.

The routing algorithm is simple: Let neighbor $n_i$ be in-charge of the range $[l_i, r_i)$. When a node is asked to route a value $v$, it chooses the neighbor $n_i$ for which $d(l_i, v)$ is minimized where $d$ denotes the *clockwise distance* or *value-distance*. Let $m_a$ and $M_a$ be the minimum and maximum values for attribute $a$, respectively.

Then,

$$d(a,b) = \begin{cases} b-a & \text{if } a \leq b, \\ (M_a - m_a) + (b-a) & \text{if } a > b \end{cases}$$

A node $n$ whose value range is $[l, r)$ constructs its long-distance links in the following fashion: Let $I$ denote the unit interval $[0, 1]$. For each link, a node draws a number $x \in I$ using the *harmonic* probability distribution function: $p_n(x) = 1/(n \log x)$ if $x \in [\frac{1}{n}, 1]$. It contacts a node $n'$ (using routing protocol itself) which manages the value $r + (M_a - m_a)x$ (wrapped around) in its hub. Finally, it attempts to make $n'$ as its neighbor. As a practical consideration, we set a fan-in limit of $2k$ links per node. We will refer to a network constructed according to the above algorithm as a ValueLink network.

[3]Self-certifying names/security are additional valuable properties.

Under the assumption that node ranges are uniform, we can prove (see [18]) that the expected number of routing hops for routing to any value within a hub is $\mathcal{O}(\frac{1}{k}\log^2 n)$. Since inter-hub routing can take at most one hop, the number of hops taken for routing[4] is at most $\mathcal{O}(\frac{1}{k}\log^2 n)$ as well. This guarantee is based upon Kleinberg's analysis of small-world networks [14].

Unfortunately, the "uniform node ranges" assumption can be easily violated for many reasons. For example, explicit load-balancing would cause nodes to cluster closely in parts of the ring which are popular. In the Section 4, we present a novel distributed histogram maintenance scheme based on light-weight random sampling to provide efficient routing even with highly non-uniform ranges.

**Caching**

For many applications, there can be significant locality in the generated data-items (incremental updates, for example) as well as queries (popular searches, for example.) Mercury provides hooks for the application so that it can insert its own specific caching behavior into the protocol. Essentially, Mercury allows an application to specify additional long-distance links that represent cached destinations as an addendum to the routing table. When looking for the neighbor closest to the destination, Mercury also considers nodes present in the cache.

### 3.4 Node Join and Leave

While the above describes the steady-state behavior of Mercury, it does not address how nodes join or leave the system. This section describes the detailed protocol used by nodes during join and departure.

Recall that each node in Mercury needs to construct and maintain the following set of links: *a)* successor and predecessor links within the attribute hub, *b)* $k$ long-distance links for efficient intra-hub routing and *c)* one cross-hub link per hub for connecting to other hubs. The cross-hub link implies that each node knows about at least one representative for every hub in the system. In order to recover during node departures, nodes keep a small number (instead of one) of successor/predecessor and cross-hub links.

*Node Join.* Like most other distributed overlays, an incoming Mercury node needs information about at least one (or at most a few) node(s) already part of the routing system. This information can be obtained via a match-making server or any other out-of-band means. The incoming node then queries an existing node and obtains state about the hubs along with a list of representatives for each hub in the system. Then, it randomly chooses a hub to join and contacts a member $m$ of that hub. The incoming node installs itself as a predecessor of $m$, takes charge of half of $m$'s range of values and becomes a part of the hub circle.

To start with, the new node copies the routing state of its successor $m$, including its long-distance links as well as links to nodes in other hubs. At this point, it initiates two maintenance processes: firstly, it sets up its own long-distance links by routing to newly sampled values generated from the harmonic distribution (as described above.) Secondly, it starts random-walks on each of the other hubs to obtain new cross-hub neighbors distinct from his successor's. Notice that these processes are not essential for correctness and only affect the efficiency of the routing protocol.

*Node Departure.* When nodes depart, the successor/predecessor links, the long-distance links and the inter-hub links within Mercury must be repaired. To repair successor/predessor links within

[4]For a query, we count the number of routing hops to reach the first value in the range it covers.

a hub, each node maintains a short list of contiguous nodes further clockwise on the ring than its immediate successor. When a node's successor departs, that node is responsible for finding the next node along the ring and creating a new successor link.

A node's departure will break the long-distance links of a set of nodes in the hub. These nodes establish new long-distance links to replace the failed ones. Nodes which are not directly affected by this departure do not take any action. The departure of several nodes, however, can distort the distribution of links of nodes which are not affected directly. To repair the distribution, nodes periodically re-construct their long-distance links using recent estimates of node counts. Such repair is initiated only when the number of nodes in the system changes dramatically (by a factor of 2 – either by addition or departure.)[5]

Finally, to repair a broken cross-hub link, a node considers the following three choices: *a)* it uses a backup cross-hub link for that hub to generate a new cross-hub neighbor (using a random walk within the desired hub) or *b)* if such a backup is not available, it queries its successor and predecessor for their links to the desired hub, or *c)* in the worst case, the node contacts the match-making (or bootstrap server) to query the address of a node participating in the desired hub.

# 4 Efficiency in the Face of Non-uniformity

The Mercury protocol we have described thus far is largely a derivative of previous structured overlay protocols. We have shown that it can provide efficient (logarithmic) routing when the responsibility of handling various attribute values is uniformly distributed to all nodes within a hub. However, as alluded to in Section 3.2, the desire to balance routing load can create a highly non-uniform distribution of ranges.

We begin this section by analyzing why such non-uniform range distributions make the design of efficient distributed routing protocols hard. We find that Kleinberg's basic small-world network result makes certain assumptions which are non-trivial to satisfy in a distributed setting when node ranges in a network are non-uniform. We then present a novel algorithm which ensures efficient routing even when the assumptions are violated.

We then tackle non-uniformity in two other dimensions: query selectivity, and data popularity. We show how the core of the algorithm for efficient routing under non-uniform range distributions can be re-used to optimize query performance given non-uniformity in query selectivity and data popularity.

## 4.1 Small-world Networks

Let $G$ represent a circle on $n$ nodes. Define *node-link distance* $d_n(a, b)$ between two nodes $a$ and $b$ as the length of the path from $a$ to $b$ in the *clockwise* direction. The objective is to find "short" routes between any pair of nodes using distributed algorithms. Kleinberg [14] showed that if each node, $A$, in $G$ constructs one additional "long-link" in a special manner, the number of expected hops for routing between any pair of nodes becomes $\mathcal{O}(\log^2 n)$. Each node $A$ constructs its link using the following rule: $A$ generates an integer $x \in (0, n)$ using the harmonic distribution, viz., $h_n(x) = 1/(n \log x)$, and establishes a link to the node $B$ which is $x$ *links away in the clockwise direction from* $A$. The routing algorithm for each node is to choose the link which takes the packet

[5]Intuitive justification: routing performance is only sensitive to the *logarithm* of the number of nodes

closest to the destination with respect to the node-link distance. Symphony [18] extends this result by showing that creating $k$ such links reduces the routing hop distance $\mathcal{O}(\frac{1}{k}\log^2 n)$.

Creating the long-links appears deceptively straight-forward. However, it may be difficult and expensive ($\mathcal{O}(x)$) for a node $A$ to determine which node, $B$, is $x$ hops away from it. Contacting node $B$ would be simpler if we could easily determine what value range $B$ was responsible for. This would allow the use of any existing long-links to contact this node more efficiently and reduce the number of routing hops to $\mathcal{O}(\log^2 n)/k$.

In systems like Symphony, this problem is solved by approximating the hop distance of any node. Since Symphony places nodes randomly along its routing hub, it makes the assumption that all nodes are responsible for approximately for ranges of the same size, $r$. By simply multiplying $r$ by $x$ and adding to the start of the values range of $A$, Symphony is able to estimate the start of the range that $B$ is responsible for. Unfortunately, this technique does not work when not all nodes are responsible for the same range size of values, i.e., when ranges are highly non-uniform in size.

In Mercury, each node maintains an approximate map of hop count to value range. It does this by sampling nodes throughout the system to determine how large a range that they are responsible for. We use these samples to create an estimate of the density of nodes in different parts of the routing hub, i.e., a histogram of the distribution of nodes. This allows us to easily map from the value $x$ to the start of the value range for $B$. This mapping, in turn, enables us to construct the long-distance links of Section 3.3 despite non-uniform node ranges. The next subsection details our techniques for uniformly sampling nodes in a distributed overlay and how they are used to maintain approximate histograms.

### 4.2 Random Sampling

Maintaining state about a uniformly random subset of global participants in a distributed network, in a scalable, efficient and timely manner is non-trivial. In the context of our system, the naïve approach of routing a sample request message to a randomly generated *data-value* works well only if node ranges are uniformly distributed. Unfortunately, as already explained, this assumption is easily violated.

Another obvious approach is to assign each node a random identifier (by using a cryptographic hash, for example) and route to a randomly generated identifier to perform sampling. However, in order for the sampling process to be efficient, we need a routing table for delivering messages to node identifiers. Another approach is to use protocols like Ransub [15] which are specifically designed for delivering random subset information. Unfortunately, both these approaches require incurring the overhead of maintaining a separate overlay – one which may not be well suited for efficient data-value routing.

Mercury's approach for sampling is novel – we show that the hub overlay constructed by Mercury in a randomized manner is an *expander* [19] with a high probability. An expander has the property that random walks over the links of such a network converge very quickly to the stationary distribution of the random walk. Since the hub overlay graph is regular, the stationary distribution is the uniform distribution. We state the lemma in a semi-rigorous manner. [6]

**Lemma.** *Let $G$ be a circle on $n$ nodes with $\mathcal{O}(\log n)$ additional links per node generated using the harmonic probability distribution (as described in Section 4.1). Let $\Pi_\infty$ denote the stationary distribution of a random walk on $G$ and let $\Pi_t$ denote the distribution generated by the random walk*

[6] The proof is omitted for reasons of space, and will be available in a related tech-report.

*after $t$ steps. Then, with high probability, $d_1(\Pi_t, \Pi_\infty) < \epsilon$ for $t > \mathcal{O}(\log^c(n/\epsilon))$ for small constants $c$, where $d_1$ denotes the statistical or $L_1$ distance between two distributions. (See [21] for rigorous definitions.)*

This leads to a very simple algorithm for performing random sampling: send off a `sample-request` message with a small (e.g., $\log n$ hop) Time-To-Live (TTL). Every node along the path selects a random neighbor link and forwards it, decrementing the TTL. The node at which the TTL expires sends back a sample. Notice that this algorithm uses only local information at every stage in the sampling process and adapts easily to a highly dynamic distributed overlay. In addition, these messages could be piggy-backed on any existing keep-alive traffic between neighbors to reduce overhead. Our simulations show that Mercury can indeed perform near-perfect uniform random sampling using a TTL of $\log n$.

We now describe three important ways in which we utilize random sampling in our system viz., to maintain node-count histograms, for estimating the selectivity of queries and for effective load balancing.

### 4.2.1 Maintaining Approximate Histograms

This section presents the mechanism used by nodes for maintaining histograms of any system statistic (e.g., load distribution, node-count distribution[7], etc.) The basic idea is to sample the distribution *locally* and exchange these estimates throughout the system in an epidemic-style protocol.

Let $N_d$ denote the "local" $d$-neighborhood of a node - i.e., the set of all nodes within a distance $d$ *ignoring the long distance links*. Each node periodically samples nodes $\in N_d$ and produces a local estimate of the system statistic under consideration. For example, if the node-count distribution is being measured, a node's local estimate is $(M_a - m_a)|N_d|/(\sum_{k \in N_d} |r_k|)$ where $r_k$ is the range of a node $k$ and $m_a, M_a$ are the minimum and maximum attribute values for the attribute $a$. In our experiments, we use $d = 3$.

In addition, a node periodically samples $k_1$ nodes uniformly at random using the sampling algorithm described in Section 4.2. Each of these nodes reports back its local estimate and the most recent $k_2$ estimates it has received. As time progresses, a node builds a list of tuples of the form: $\{$`node_id`, `node_range`, `time`, `estimate`$\}$. (The timestamp is used to age out old estimates.) Each of these tuples represent a point on the required distribution – stitching them together yields a piecewise linear approximation.

$k_1$ and $k_2$ are parameters of the algorithm which trade-off between overhead and accuracy of the histogram maintenance process. In Section 5, we show through simulations that setting each of $k_1$ and $k_2$ to $\log(n)$ is sufficient to give reasonably accurate histograms for sampling population distribution.

If the system needs to generate an average or histogram of node properties, the collected samples can be used exactly as they are collected. However, if the desire is to generate an average or histogram of properties around the routing hub, some minor modifications are needed.

Let us consider our objective of estimating the number of nodes in any part of the hub. Each sample represents a node density estimate for a point on the hub. However, we must decide where this estimate is valid. A simple solution would be to say that a sample is valid up to the location half-way between the sample's origin point and origin of the next sample along the hub.

[7]Number of nodes responsible for a given range of values.

For example in a unit hub, let us consider a situation where we have two samples: one at 0.25 measuring a density of 10 nodes in the system and one at .75 with a 40 node measurement. We could say that half the hub (range 0.0-0.5) has 5 nodes (10 divided by 2) and the other half has 20, for a total of 25 nodes. However, this introduces a systematic bias against low-density samples since there are obviously less nodes that produce them.

To correct for this, we make a sample valid up to a mid-point that is weighted by the relative densities of the neighboring samples. For the above situation, we would say that there is a density of 10 in the range 0.85-0.65 (wrapping around clockwise), and a density of 40 between 0.65 and 0.85. This would result in a total of 8 (10 * .8) in the low density region and 8 (40 * .2) in the high density region for a total of 16 nodes. We have performed experiments that show that this produces a much higher accuracy estimate. Given such a density estimate for different parts of the hub, we can easily generate the data needed to efficiently contact neighbors for constructing a small-world graph.

### 4.3 Query Selectivity

Recall that a query $Q$ is sent to only one of the attribute hubs in $\mathcal{A}_Q$. Also a query $Q$ is a *conjunction* of its predicates each of which can have varying degree of selectivity. For example, some predicate might be a *wildcard* for its attribute while another might be an exact match. Clearly, a wildcard predicate will get flooded to every node within its attribute hub. Thus, the query $Q$ should be sent to that hub for which $Q$ is *most selective* to minimize the number of nodes that must be contacted.

The problem of estimating the selectivity of a query has been very widely studied in the database community. The established canonical solution is to maintain approximate histograms of the number of database records per bucket. In our case, we want to know the number of nodes in a particular bucket. Each node within a hub can easily gather such an histogram for its own hub using the histogram maintenance mechanism described above. In addition, using its inter-hub links, it can also gather histograms for other hubs efficiently. These histograms are then used to determine the selectivity of a subscription for each hub. We see in Section 5.4 that even with a very conservative workload, this estimation can reduce a significant amount of query flooding.

### 4.4 Data Popularity and Load Balancing

When a node joins Mercury, it is assigned responsibility for some range of an attribute. Unfortunately, in many applications, a particular range of values may exhibit a much greater popularity in terms of database insertions or queries than other ranges. This would cause the node responsible for the popular range to become overloaded. One obvious solution is to determine some way to partition the ranges in proportion to their popularity. As load patterns change, the system should also move nodes around as needed.

We leverage our approximate histograms to help implement load-balancing in Mercury. First, each node can use histograms to determine the average load existing in the system, and, hence, can determine if it is relatively heavily or lightly loaded. Second, the histograms contain information about which parts of the overlay are lightly loaded. Using this information, heavily loaded nodes can send probes to lightly loaded parts of the network. Once the probe encounters a lightly loaded node, it requests this lightly loaded node to gracefully leave its location in the routing ring and re-join at the location of the heavily loaded node. This leave and re-join effectively increases the

load on the neighboring (also likely to be lightly-loaded) nodes and partitions the previous heavy load across two nodes.

Let the average load in the system be denoted by $\bar{L}$. Define the *local load* of a node as the average of load of itself, its successor and its predecessor. A node is said to be lightly loaded if the ratio of its local load to $\bar{L}$ is less than $\frac{1}{\alpha}$ and heavily loaded if the ratio is greater than $\alpha$. This definition ensures that if a node is lightly loaded, its neighbors will be lightly loaded with a high probability. If this is not the case (when the ratio of neighbor loads is $> \alpha$), the lighter neighbor performs a load balance with the heavier one to equalize their loads. It is easy to show[8] that the leave-rejoin protocol described above *decreases* the variance of the load distribution at each step and bounds the maximum load imbalance in the converged system by a factor of $\alpha$, provided $\alpha \geq \sqrt{2}$. By tolerating a small skew, we prevent load oscillations in the system.

Over time, the leaves and re-joins result in a shift in the distribution of nodes to reflect the distribution of load. However, this shift in node distribution can have significant implications. Many of the properties of Mercury's routing and sampling rely on the harmonic distance distribution of the random long-links. When nodes move to adjust to load, this distribution may be changed. However, our technique for creating long-links actually takes the node distribution into account explicitly as stated previously.

We emphasize that this load balancing mechanism (leave-join) is not itself new; similar techniques have been proposed [13,22]. Our novel contribution here is the random sampling mechanism which *enables* the use of such techniques in distributed overlays when node range distributions are skewed.

# 5 Evaluation

This section presents a detailed evaluation of the Mercury protocol using simulations. We implemented a simple discrete event-based simulator which assigns each application level hop a unit delay. To reduce overhead and enable the simulation of large networks, the simulator does not model any queuing delays or packet loss on links. The simplified simulation environment was chosen for two reasons: first, it allows the simulations to scale to a large (up to 50K) number of nodes, and secondly, this evaluation is not focused on proximity routing. Since our basic design is similar in spirit to Symphony and Chord, we believe that heuristics for performing proximity-based routing (as described in [10]) can be adapted easily in Mercury.

Our evaluation centers on two main features of the Mercury system: *1)* scalable routing for queries and data-records, and *2)* balancing of routing load throughout the system. We begin with an evaluation of our core routing mechanisms – random sampling and histogram maintenance. We then study the impact of these mechanisms on the overall routing performance under various workloads. Finally, we present results showing the utility of caching and query selectivity estimation in the context of Mercury.

Except for query selectivity estimation, most of our experiments focus on the routing performance of data *within* a single routing hub. Hence, $n$ will denote the number of nodes within a hub. Unless stated otherwise, every node establishes $k = \log n$ intra-hub long-distance links. For the rest of the section, we assume without loss of generality that the attribute under consideration is a `float` value with range $[0, 1]$. Each node in our experiments is thus responsible for a value interval

[8] We omit the proof for reasons of space. The idea is simply that variance reduction 'near' the heavier node is larger than the variance increase 'near' the lighter node.

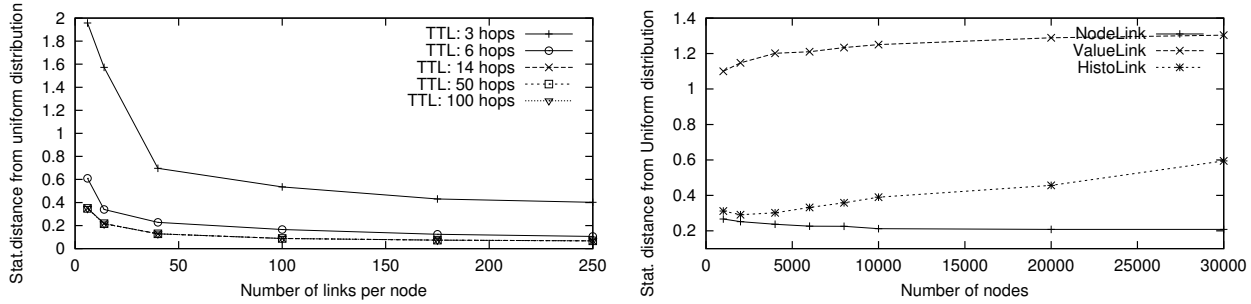



**Figure 3: Accuracy of random-walk based sampling. Figure (a) shows the effect of the degree of the graph. ($n = 10000; \log n = 14$; NodeLink overlay.) Figure (b) shows the effect of link structure.**

$\subset [0, 1]$.

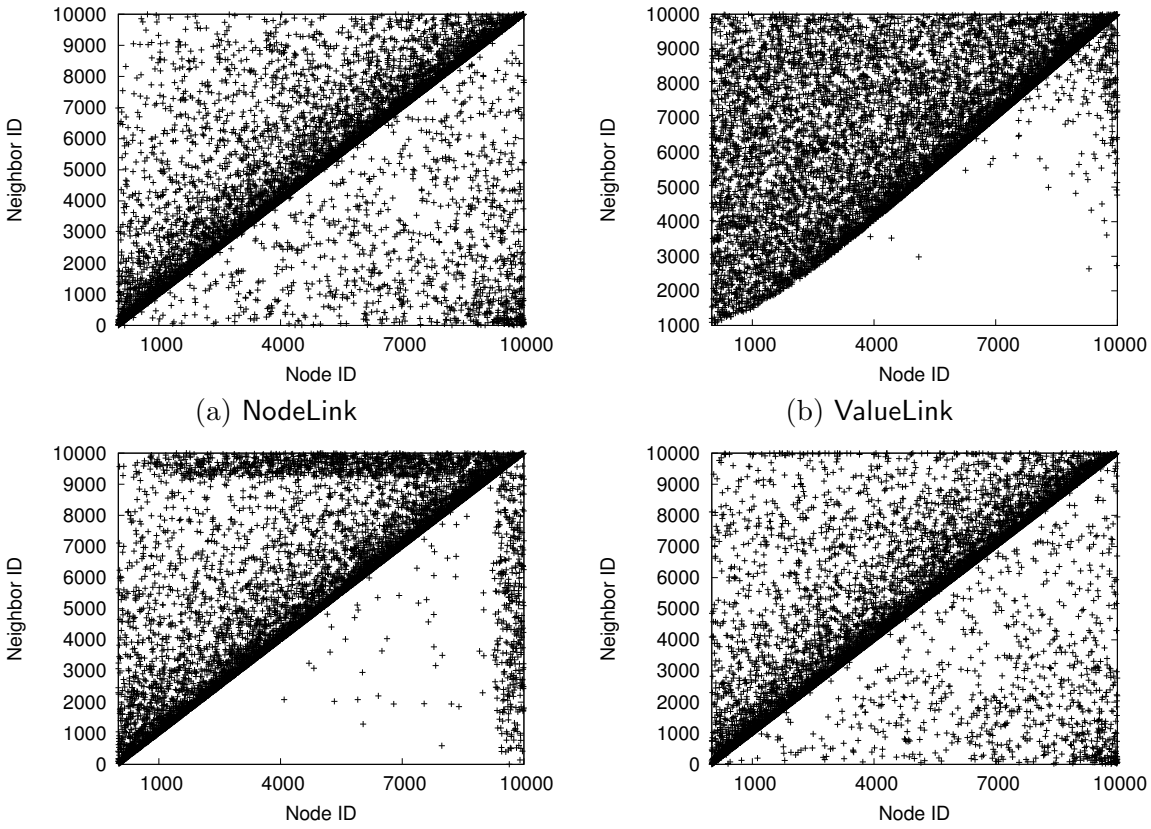
In what follows, NodeLink denotes the ideal small-world overlay, i.e., long distance links are constructed using the harmonic distribution on node-link distance. ValueLink denotes the overlay when the harmonic distribution on value-distance is used (Section 3.3). HistoLink denotes the scenario when links are created using node-count histograms (see Section 4.) Note that the performance of the ValueLink overlay is representative of the performance of a plain DHT (e.g., Chord, Symphony) under the absence of hashing and in the presence of load balancing algorithms which preserve value contiguity.

For evaluating the effect of non-uniform node ranges on our protocol, we assign each node a range width which is inversely proportional to its popularity in the load distribution. Such a choice is reasonable since load balancing would produce precisely such a distribution – more nodes would participate in a region where load is high. The ranges are actually assigned using a Zipf distribution. In particular, data values near 0.0 are most popular and hence a large number of nodes share responsibility for this region, each taking care of a very small node range. For reference, in our simulator setup, these are also the nodes with lowest numeric IDs.

## 5.1 Random-Walk Based Sampling

The goal of our random-walk based sampling algorithm is to produce a uniform random sample of the nodes in the system. We measure the performance of our algorithm in terms of the statistical distance (alternatively called $L_1$ distance) of the perfect uniform distribution from the distribution obtained via the random walks. For these experiments, nodes are assigned ranges using a highly-skewed Zipf distribution. In each sampling experiment, we pick a node at random and record the distribution of the samples taken by $kn$ random walks starting from this node. If our sampling algorithm is good, the random walks should hit each node roughly $k$ times. Note that the parameter $k$ is just for evaluating the distribution obtained – the protocol does not use it in any manner.

Figure 3(a) plots the accuracy of the sampling process as the degree of the graph and the TTL for the random-walks is varied. The underlying overlay we consider is a perfect small-world network (NodeLink). We find that, over a certain threshold ($\log n$), the TTL of the random-walks does not influence sampling accuracy. Also, the sampled distribution is almost perfectly random for graph degrees $c \log n$, where $c$ is a small constant. In practice, we found that, for routing purposes, sufficiently accurate histograms are obtained even for $c = 1$.




(a) NodeLink (b) ValueLink

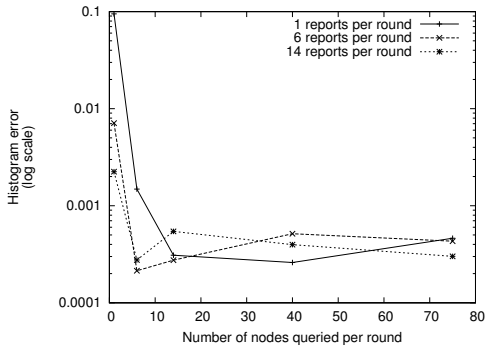
(c) HistoLink [1:1] (d) HistoLink $[\log n : \log n]$

**Figure 4: Distribution of long-distance links. The Y-axis plots the ID of the $k = \log n$ neighbors for each node on the X-axis. Nodes are consecutively arranged on the hub circle. Number of nodes = 10000. For HistoLink, $[k_1 : k_2]$ means $k_1$ nodes were queried per round each giving $k_2$ estimate reports; 5 exchange rounds were performed.**

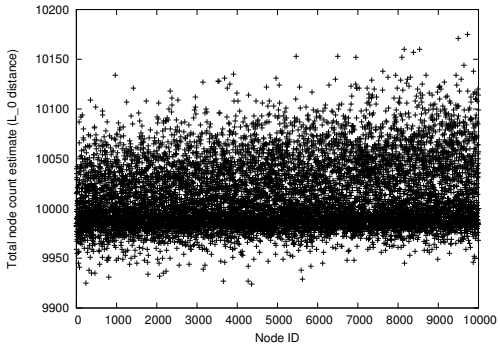
Figure 3(b) shows how the construction of the underlying network affects sampling accuracy. We see that the NodeLink and HistoLink overlays perform much better than the ValueLink (a vanilla DHT without hashing and in the presence of load balancing) overlay. These effects are explained using Figure 4 which plots the distribution of long-distance links. As described earlier, in our experiments, nodes with the lowest identifiers (responsible for values near 0.0) are the most popular while nodes at the other end of the value range are the least popular.

Recall that, in a ValueLink overlay, nodes construct their links by routing to *values* generated using a harmonic distribution. However, in this case node ranges are not uniformly distributed – in particular, nodes near the value 1.0 (i.e., nodes with higher IDs) are less popular, they are in charge of larger range values. Hence, the long-distance links they create tend to *skip* over less nodes than appropriate. This causes all the links (and correspondingly, the random walks) to crowd towards the least popular end of the circle. The HistoLink overlay offsets this effect via the maintained histograms and achieves sampling accuracy close to that achieved by the optimal NodeLink overlay.

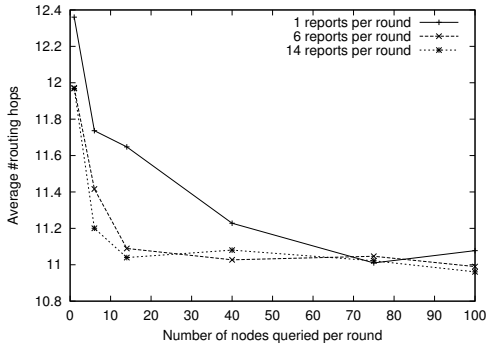
Each `sample-request` message travels for TTL hops and hence obtaining one random sample generates TTL additional messages in the overall system. However, all these messages are sent over existing long-distance links. Thus, they can be easily multiplexed and piggy-backed (by simply




(a) Histogram error ($L_1$-distance)




(b) Node count estimates




(c) Effect on routing performance

**Figure 5: Accuracy of sampled histograms.** ($n = 10000$) **Figure (a) shows the node count estimates gathered by each node. Figure (b) shows the effect of changing parameters on average histogram error. Figure (c) shows effect of changing parameters on overall routing performance.**

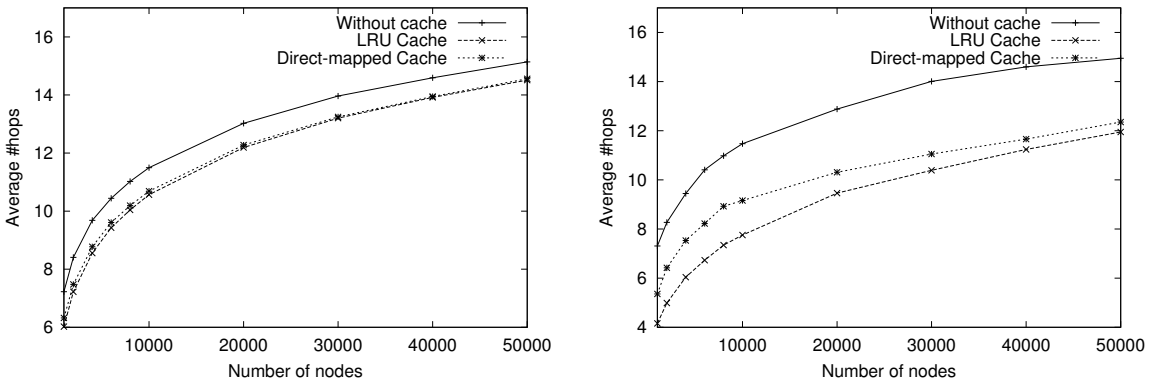
appending the IP address of the requesting node) over the regular keep-alive pings that might be sent between neighbors. Also, if the samples are uniformly distributed over all nodes, each node receives $\mathcal{O}(1)$ sample requests per sampling period. Hence, we conclude that the overhead of the sampling method is very small.

## 5.2 Node-Count Histograms

In this section, we evaluate the accuracy of the node-count histograms obtained by nodes using the mechanism described in Section 4.2. These histograms, introduced in Section 4.2.1, provide an estimate of the total number of nodes in the system and help in establishing the long-distance links correctly.

We measure the accuracy of the obtained histogram in terms of its *distance* from the "true" histogram under the $L_1$ norm. Figure 5(a) plots the average accuracy of the histogram (the average is taken over all nodes) as the parameters for the histogram maintenance process are varied. In this experiment, 10 rounds of exchanges were performed. We see that the error is consistently small and decreases rapidly as the number of the nodes queried per round increases.[9] Although not obvious from the graph, the same pattern is observed when the number of reports queried from each node

[9]The graph does show some fluctuations, but their magnitudes are tiny (result of experimental variations).




(a) Uniform workload (b) Zipf-skewed workload

**Figure 6: Performance of the basic Mercury protocol for various workloads.**

is increased. These effects are expected, since the number of samples received by each node per round grows linearly with either of these parameters.

Figure 5(b) plots the overall node-count estimates produced by each node in a system of $n = 10000$ nodes. The experiment was run for 10 exchange rounds, with $k_1 = k_2 = \log n$. We see that the estimates are very tightly clustered around the actual node-count value of 10000.

During each round of the histogram maintenance process, each node queries $k_1$ randomly sampled nodes and receives $k_2$ estimate samples from each node. The messaging overhead per round per node is thus proportional to $k_1 k_2$.

### 5.3 Routing Performance

We now present an evaluation of the overall routing performance of Mercury. This factors in the effects of the random sampling and histogram maintenance strategies. We present the performance of the basic protocol with caching optimizations, discuss the effect of skewed node-range distributions and validate our claim that the protocol using histograms achieves near-optimal routing delays. As before, we concentrate on routing within a single hub. In each of the following experiments, nodes establish $k = \log n$ long-distance links within a hub.

We experiment with two different data workloads – *uniform* and *Zipf*. The Zipf workload is high-skewed and is generated using the distribution $x^{-\alpha}$ where $\alpha = 0.95$. Notice that this means that the attribute values near 0.0 are the most popular and those around 1.0 are the least popular. We also show the performance of two types of caching policies, viz., LRU replacement and a direct-mapped cache.[10] Our objective here is not to find the best possible policy for our workload. Rather, our aim is to show the ease with which application-specific caching can co-exist fruitfully with Mercury routing. In our caching experiments, each node keeps a cache of $\log n$ entries.

Figure 6 shows the performance of Mercury when node ranges are uniformly distributed. The Y-axis shows the average number of hops taken by a data-item to reach its destination (node where it is stored) in the hub. Although these graphs show results for HistoLink overlay, the performance of NodeLink and ValueLink is very similar, as expected.

We see that, for uniform node ranges, the number of routing hops scales logarithmically (with very low constant factors) as the number of nodes increases, irrespective of the workload used.

[10]For an $n$-entry cache, there is one entry for each of the $(1/n)^{\text{th}}$ region of the attribute space.

Thus, Mercury can provide low end-to-end routing delays to applications even for a large number of nodes. With caching enabled, there is a significant reduction in hop count. While this is easy to see for a skewed workload, the reduction for a uniform workload results from the fact that a cache effectively increases Mercury's routing table size. We believe that caching is an important optimization which Mercury can easily incorporate into its basic protocol.

**Effect of Non-Uniform Ranges**

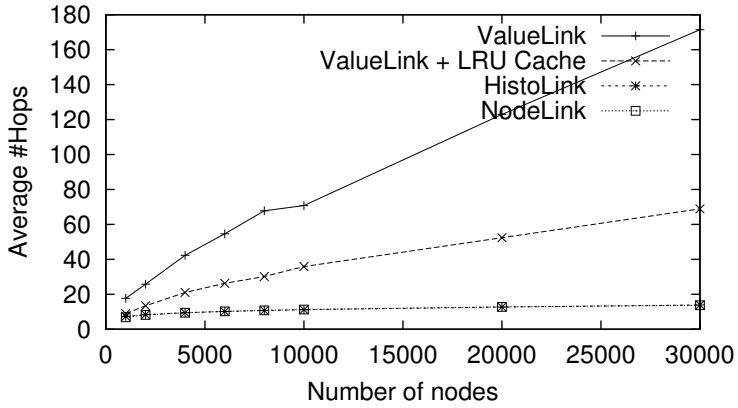



**Figure 7: Effect of non-uniform node ranges on the average number of routing hops. As workload, we use the Zipf distribution with $\alpha = 0.95$.**

Figure 7 compares the performance of the protocol with and without approximate histograms to guide the selection of the long-distance links. In this experiment, the node-range distribution and the data distribution are Zipf-skewed. For histogram maintenance in this experiment, we used 5 exchange rounds, where each node queried $\log n$ nodes per round asking each for $\log n$ estimate reports.

As explained in Section 5.1, the naïve ValueLink overlay (vanilla DHT in the presence of load balancing) creates links which *skip* the crowded and popular region (see Figure 4.) Hence, packets destined to these nodes take circuitous routes along the circle rather than taking short cuts provided by the long-distance links. Although caching ameliorates the effect, the performance is still much worse as compared to the optimal NodeLink overlay.

On the other hand, we see that the performance of the HistoLink overlay is nearly the same as that of the optimal NodeLink overlay. Again, looking at Figure 4, we find that node-count histograms enable nodes to establish a correct link distribution (corresponding to the NodeLink overlay) quickly using very low overheads.

Figure 5(c) shows the effect of histogram accuracy on the overall routing performance. We see that as the parameters $k_1$ and $k_2$ in the histogram maintenance process increase, the routing performance improves as expected. We note that this influence is limited (note the scale of the graph) since it is directly dependent on the accuracy of the obtained histograms (see Figure 5(a).)

## 5.4 Estimating Query Selectivity

To evaluate the usefulness of forwarding queries to the most selective attribute hubs, we set up an experiment with 3 attribute hubs. Our workload is motivated by the distributed multi-player game

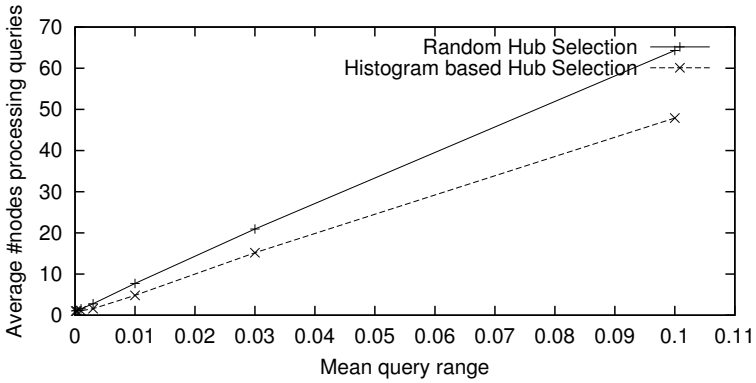



**Figure 8: Network bandwidth consumption for forwarding queries.**

application we describe in Section 6. The attribute hubs correspond to three dimensions of virtual space. Each query contained 3 range predicates – one for each attribute. Such a query specifies a cuboid region of the virtual space. The range-size of each predicate was Gaussian-distributed, while the range position within the attribute space was Zipf-distributed. The node-range distribution within each hub is skewed.

Figure 8 plots the average number of nodes processing the query as the number of nodes grows. As the number of nodes grows, the range of values each node is responsible for decreases. Hence, each query tends to get more flooded within each hub. The plot shows that, even within this conservative setting, selecting a hub based on the selectivity estimates results in up to 25-30% reduction in the degree of flooding of a query. With workloads exhibiting wildcards, much higher reductions would be expected.

### 5.5 Load Balancing

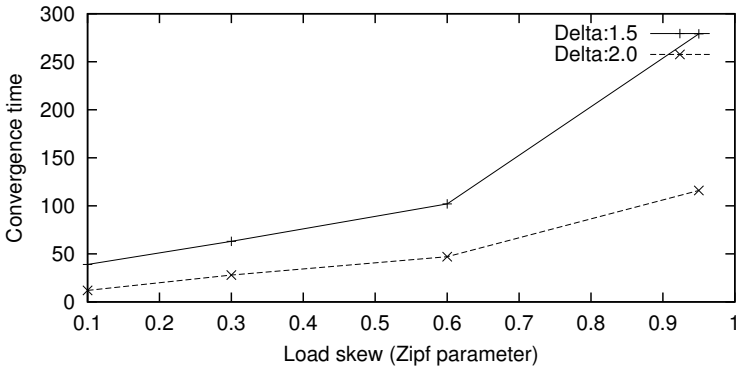



**Figure 9: Graph showing rounds taken to achieve load balance as a function of the initial skew of the load. $\Delta$ is the degree of load balance sought.**

For evaluating the efficiency of load balancing achieved by Mercury's load balancing algorithm, we conduct the following experiment: In a system of 1000 nodes, we assign load to each node using a Zipf distribution with varying values of the initial skew (Zipf parameter). The system is said to be load-balanced when $\frac{1}{\Delta} \leq load/avg\_load \leq \Delta$ holds for all nodes in the system.

In the experiment, we run multiple *rounds* of the load balancing algorithm, until the system is load-balanced. Each round consists of the following steps:

1. Each node samples its neighbors, to determine the local node-count. This requires one round-trip.
2. Each node runs one round of the histogram maintenance algorithm. (Recall that a round of the histogram maintenance algorithm involves sending $\log n$ probes in parallel, each of which must traverse $1 + \log n$ hops.)
3. Nodes check their histograms to determine if they are heavily loaded. If a node is heavily loaded, it sends a probe to a lightly loaded node. This probe must traverse $\log n$ hops.
4. Lightly loaded nodes leave and re-join the network. To re-join, the lightly loaded nodes must establish new long links. The link establishment messages traverse $1 + \log n$ hops, in expectation.

Figure 9 plots the number of rounds of load-balancing required to achieve load balance. We see that Mercury can load-balance to within a factor of $\Delta = 2$ within 100 rounds despite heavy skews in the workload (Zipf with $\alpha = 0.95$). In practical terms, consider an overlay with 10000 nodes, and a 50 ms delay between nodes. The time to complete one round of load-balancing is the product of the number of hops traversed by messages in the load balancing algorithm [11], and the inter-node delay. Thus the time to complete one round is $50 * (4 + 3 \log n)$ ms. The time to load-balance the entire overlay is then $100 * 50 * (4 + 3 \log n)$ ms, or about 220 seconds.

# 6 Distributed Object Management

Previous sections have demonstrated that Mercury provides scalable range-query based lookups under a variety of workloads. In this section, we describe how Mercury can also be used as a *building block* for many distributed applications. We present a distributed object management framework based on Mercury, followed by the design and testbed evaluation of a distributed multiplayer game called *Caduceus* built on top of this framework.

## 6.1 Modeling Object Interactions

One of the challenges of distributed application design is managing distributed state, which includes *code objects and functions*, partitioned across multiple, perhaps physically distributed, machines. In such an environment, performing object lookups, updates and maintaining consistency in a scalable manner is difficult.

We observe that, in many applications, each instance is typically interested in only a small subset of the entire application state. Moreover, the objects belonging to this subset are related to each other in an application-specific manner. This observation enables the following publish-subscribe [3] architecture: each application instance registers a "subscription" describing the objects which it wishes to keep updated. When an object is updated, these updates act as "publications" and are delivered to the interested instances.

[11] Since the messages in step 2 are sent in parallel, we count the number of hops once, rather than multiplying by the number of messages. Similarly for step 4

A key requirement in this design is a flexible subscription language which allows the application to express its object-interests precisely. At the same time, the language should also permit a scalable implementation for the underlying publish-subscribe routing infrastructure. By providing scalable routing for a multi-attribute range-query based subscription language, Mercury fits in this need very well.

While the previous sections have demonstrated Mercury's desirable performance characteristics, a high performance system is not useful unless it meets the needs of real applications. Thus, to illustrate the usefulness of Mercury, we have implemented the Mercury publish-subscribe system as a C++ library, and used this library (`libmercury`) to build a distributed multiplayer game (*Caduceus*).

Given that we focus on a single application, a natural question is how well our system generalizes. As we have not yet implemented other applications, we cannot provide empirical evidence of generalizability. However, we believe our system provides a general API that will be suitable for many applications.

### 6.2 Caduceus

Caduceus is a two-dimensional, multi-player, shooter game. Each player in the game has a ship and a supply of missiles. Players pursue each other around the two-dimensional space, and fire missiles when their opponents are in range. The goal is simply to kill as many opponents are possible. Figure 10 presents a screenshot of the game. At any given time, a player sees the other ships in her immediate vicinity, as defined by the game window. The virtual game world is larger than the window. Thus there might, for example, be opponents located beyond any of the edges of the game window.

The state of the game is represented using two kinds of objects: ships and missiles. A ship consists of a location, velocity, and ancillary information such as fuel level. A missile is created whenever a ship fires shots. A missile consists of a location, velocity, and owner information [12]. The main loop of Caduceus, shown in Figure 11, is relatively simple.

### 6.3 Messaging Architectures for Distributed Games

In multiplayer games, such as Caduceus, multiple users exist in a single game "world". The world contains a number of objects, such as people, ships, weapons, obstacles, etc. A central problem in *distributed* multiplayer gaming is ensuring that all nodes have consistent views of the game world. To provide this consistency, nodes send updates to other nodes whenever the world's state has changed. To date, most games have used either broadcast messaging, or a centralized server, to deliver these updates.

In broadcast based games (such as Doom and MiMaze [8]), each node broadcasts every change in the state of the game world (such as a player moving from one position to another) to all other nodes. Broadcasting limits scalability in two ways. First, it imposes high load on the network. Second, it forces nodes to process updates that may be irrelevant to them (e.g., movements in a distant part of the world).

Centralized designs (e.g., Quakeforge [20]) improve upon broadcast messaging by filtering updates at a game server. In these designs, every node sends changes in the world state to a central server. The game server sends only relevant updates (e.g. movements of players in the immediate

[12]The owner is used to credit the appropriate player when an opponent is killed.

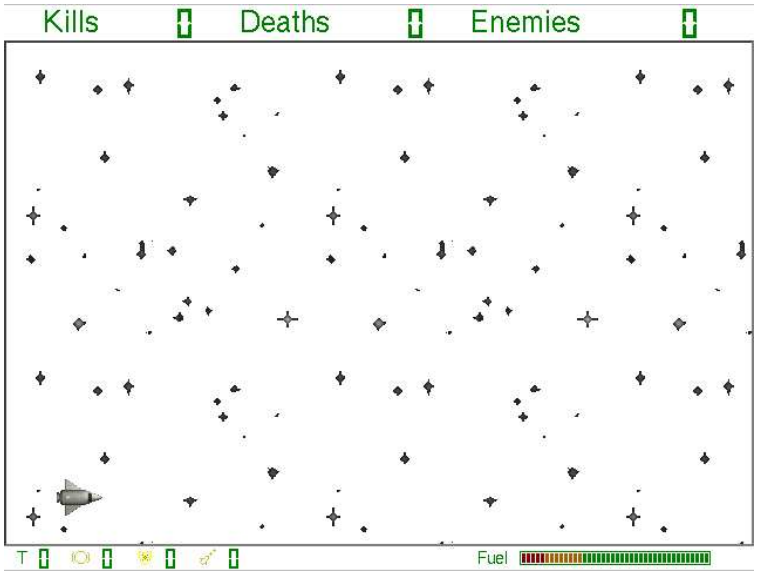



**Figure 10: Caduceus screenshot**

```
void GameApp::timerEvent(QTimerEvent *) {
  m_Renderer->Render();
  m_GameEngine->RunFrame(); // read keyboard events,
                            // run physics
  updateSubs();
  m_StateManager->UpdateState(); // publish dirty objects,
                                 // receive pubs
  m_StateManager->CollectGarbage(); // delete useless objects
}
```

**Figure 11: Main loop of Caduceus**

vicinity) to each node. These designs suffer, however, from high network and computational loads at the central server.

To improve scalability, researchers have proposed area-of-interest filtering [17, 25] schemes. In these schemes, the game world is divided into a fixed set of regions (such as tiles of a two-dimensional space). The regions are then mapped to IP multicast groups. Such approaches improve scalability by distributing the message filtering throughout the network. However, the fixed regions result either in the delivery of a large number of irrelevant updates to clients, or in the maintenance of a large number of IP multicast groups at routers.

In contrast, Mercury's subscription language is ideal for implementing area of interest filtering. In particular, the subscription language makes it easy to describe arbitrary physical regions. As an example, Figure 12 shows two nodes expressing their interests in the rectangular regions near them. Of interest is the fact that the regions do not, for example, need to fit a pre-defined tiling of the space. Note that while tiling the space, and assigning these tiles to different channels, would be possible for a simple two-dimensional game, it becomes far more difficult in games with irregular spaces, such as corridors, or which have to deal with visibility constraints such as horizons. It is, of course, also difficult for channel-based schemes to support arbitrary interests such as "the location

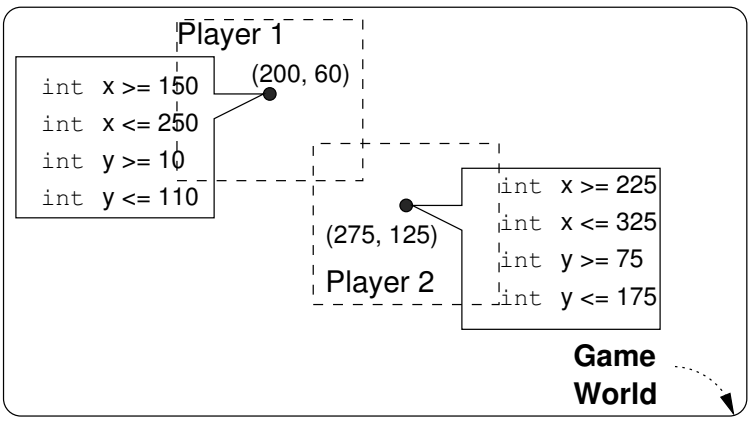



**Figure 12: Example subscriptions for Caduceus**

of all my teammates".

As shown in Figure 6(b), Mercury is able to handle 10000 nodes while keeping the number of routing hops below 8. Assuming that the average-case, one-way delay between nodes is about 20ms (e.g., they are all well connected and within the U.S. west coast), this results in an end-to-end delay of less 160ms. We believe that game-specific caching algorithms could further improve the routing performance, making Mercury-based games scalable to thousands of nodes.

### 6.4 Managing Distributed Objects

As noted in Sec. 6.3, a distributed game must ensure that all nodes have consistent views of the game world. Mercury provides a solution for two aspects of that problem — specifying the areas of interest for game nodes, and efficiently routing world state updates to the relevant nodes. However, a complete solution requires additional support. In particular, the game requires facilities for applying state updates to its local state, for creating update messages that reflect local changes, for enumerating the local state, and for managing update conflicts.

Having identified these needs, we now describe our *object management framework*, which bridges the gap between the services provided by Mercury, and the services required by distributed games. The object management framework, implemented as a C++ library (`libframework`), interfaces with our game application via one fundamental data structure, several API calls, and two upcalls. We describe these in turn.

The fundamental data structure of the framework is the *object registry*. The object registry is a collection of all the objects relevant to the local view of the game. The framework ensures that any remote changes to objects in the registry are reflected locally, and vice versa. Objects to be placed in the registry must inherit from `RNObject`, which is defined in `libframework`.

The API provides calls for interest management (expressing and cancelling interests), object management (adding and removing objects from the registry), and object updating (a method to mark objects as dirty, and a method to synchronize the local state with global state). The framework requires the application to provide three upcalls (via virtual methods of `RNObject`). These upcalls, which all relate to serialization, are: a constructor (to create local copies of remote objects), an update method, which updates a local object's state based on publications, and a serialization method.

The API calls and upcalls are sufficient to handle all the application needs we enumerated, except for update conflicts. Given that multiple nodes may have copies of the same object, it is possible that multiple nodes modify the object. To ensure that object state does not become inconsistent, the framework permits only a single writer for each object. The framework maintains a replica tag for each object, signifying whether the object was created locally, or was received from the network. When the application calls the synchronization method, the framework does not generate updates for objects that have the replica tag set. While we believe the simplicity of this model is valuable, we acknowledge that we may need to support alternate consistency models in the future.

### 6.5 API Evaluation

We evaluate the framework's API by the number of lines of code in Caduceus that relate to the functionality provided by `libframework`. We divide the costs into five groups: Interest Management, Object Management, Object Updates, Serialization, and Write Conflicts. For each group, we give the number of lines required to call the corresponding functions, as well as to prepare the relevant arguments. As an example, "Interest Management" includes the calls to the interest management method, and the code required to compute the game's current area of interest. As a point of reference, the entire Caduceus code is 3043 lines. Table 1 gives the results.

| Function Group | Lines |
|---|---|
| Interest Management | 289 |
| Object Management | 12 |
| Object Updates | 10 |
| Serialization | 170 |
| Write Conflicts | 0 |

**Table 1: Lines of code required to use `libframework`**

The most significant costs are those for interest management, and serialization. The interest management costs are due primarily to the logic of determining the application's area of interest, and would thus be required for any non-broadcast system. Broadcast systems clearly have lower programming complexity with respect to interest management. With respect to serialization code, we note that serialization code is an inherent cost of distributed applications[13]. Moreover, toolkits are available to ease the programming burden of serialization. The unique costs of using `libframework` (to interface with its object management, object update, and conflict management facilities), are quite modest. Thus, the object management framework and Mercury are indeed appropriate building blocks for a distributed game such as Caduceus.

### 6.6 Performance Evaluation

We evaluate the performance of our system with two metrics: hop count, and message count. We run a varying number of players. The players move through the world according to a random waypoint model, with a motion time chosen uniformly at random from (1, 10) seconds, a destination chosen uniformly at random, and a speed chosen uniformly at random from (0, 360) pixels per second. The

[13]It is possible to architect the game so that it sends updates which describe the changed fields, rather than serializing the entire object. However, it is not clear that doing so would reduce code complexity.

size of the game world is scaled according to the number of players. The dimensions are $640n \times 480n$, where $n$ is the number of players. All results are based on the average of 3 experiments, with each experiment lasting 60 seconds. The experiments include the benefit of a $\log n$ sized LRU cache at each node, but do not include the benefits of *any long pointers*.

Table 2 summarizes the results. With respect to hop count, we find that the hop count increases only slightly as we double the number of nodes. To evaluate Mercury's messaging efficiency, we compare it to two alternatives. In the "broadcast messages" column of the table, we report the number of messages that would have been transmitted if every update were sent to every node (as was done in first-generation distributed games). In the "optimal messages" column, we report the number of messages required to exactly satisfy each node's interests, without any control message overhead. We find that Mercury performs substantially better than a broadcast scheme (43% as many messages transmitted for 20 nodes), and that this performance difference increases when we increase the number of nodes, with Mercury using only 29% as many messages as broadcast for 40 nodes.

| # of Players | Average Hops | Broadcast Messages | Mercury Messages | Optimal Messages |
|---|---|---|---|---|
| 20 | 4.44 | 170000 | 74295 | 28154 |
| 40 | 4.61 | 695240 | 199076 | 58644 |

**Table 2: Mercury routing overheads for Caduceus, *without long pointers*.**

## 7 Conclusion

In this paper, we have described the design and implementation of Mercury, a scalable protocol for routing multi-attribute range-based queries. Our contributions as compared to previous systems include support for multiple attributes and explicit load balancing. Mercury incorporates novel techniques to support the random sampling of nodes within the system. This random sampling enables a number of light-weight approaches to performing load-balancing, node count estimation and selectivity estimation. Our evaluation clearly shows that Mercury scales well, has low lookup latency and provides good load balancing.

In addition to providing high query-routing performance, Mercury provides an easy-to-use facility for the maintenance of a distributed object database. In particular, an application using our system needs only compute its interests, provide methods for serialization and deserialization, register objects, and set a dirty tag on any modified object. By providing a range-based query language, Mercury allows applications to express their interests in a more flexible manner. While we have only directly shown the ease-of-use of Mercury for distributed games, we believe that the classes of applications that will benefit from our system include collaborative applications, such as shared whiteboards, distributed inventories and possibly sensing applications as well.

## References

[1] Byers, J., Considine, J., and Mitzenmacher, M. Simple load balancing for distributed hash tables. *Second International Workshop on Peer-to-Peer Systems* (2003).

[2] Cabrera, L. F., Jones, M. B., and Theimer, M. Herald: Achieving a Global Event Notification Service. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems* (Elmau, Germany, May 2001).

[3] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems 19*, 3 (Aug. 2001), 332–383.

[4] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. Exploiting network proximity in distributed hash tables. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)* (June 2002), O. Babaoglu, K. Birman, and K. Marzullo, Eds., pp. 52–55.

[5] Castro, M., Druschel, P., Kermarrec, A. M., Nandi, A., Rowstron, A., and A., S. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proceedings of the 19th Symposium on Operating System Principles* (Oct. 2003).

[6] Castro M., et. al. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) 20*, 8 (Oct. 2002).

[7] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th Symposium on Operating System Principles* (Chateau Lake Louise, Banff, Canada, Oct. 2001).

[8] Gautier, L., and Diot, C. Design and Evaluation of MiMaze, a Multi-player Game on the Internet. In *IEEE Multimedia Systems Conference* (Austin, TX, July 1998).

[9] Ghosh, B., Leighton, F. T., Maggs, B. M., Muthukrishnan, S., Plaxton, C. G., Rajaraman, R., Richa, A. W., Tarjan, R. E., and Zuckerman, D. Tight analyses of two local load balancing algorithms. In *Proceedings of the 27th ACM STOC* (1995), pp. 548–558.

[10] Gummadi, K. P., et. al. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of the ACM SIGCOMM '03* (Aug. 2003).

[11] Harvey, N. J. A., Jones, M. B., Saroiu, S., Theimer, M., and Wolman, A. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, Mar. 2003).

[12] Heubsch, R., Hellerstein, J., Lanhan, N., Loo, B. T., Shenker, S., and Stoica, I. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large DataBases* (Sept. 2003).

[13] Karger, D., and Ruhl, M. Simple efficient load-balancing algorithms for peer-to-peer systems. *Third International Workshop on Peer-to-Peer Systems* (2004).

[14] Kleinberg, J. The Small-World Phenomenon: An Algorithmic Perspective. In *Proceedings of the 32th ACM STOC* (2000).

[15] Kostic, D., Rodriguez, A., Albrecht, J., Bhirud, A., and Vahdat, A. Using random subsets to build scalable network services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, Mar. 2003).

[16] Li, X., Kim, Y.-J., Govindan, R., and Hong, W. Multi-dimensional range queries in sensor networks. In *Proceedings of the ACM Sensys 2003* (Nov. 2003).

[17] Macedonia, M. R., Zyda, M. J., Pratt, D. R., Brutzman, D. P., and Braham, P. T. Exploting reality with multicast groups: A network architecture for large-scale virtual environments. In *Proc. of the 1995 IEEE Virtual Reality Symposium (VRAIS95)* (Mar. 1995).

[18] Manku, G., Bawa, M., and Raghavan, P. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, Mar. 2003).

[19] Motwani, R., and Raghavan, P. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] QuakeForge. `http://www.quakeforge.net`.

[21] Randall, D. Math 8213A - Rapidly Mixing Markov Chains. http://www.math.gatech.edu/∼randall/Course/lewis1.ps, 2003.

[22] Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., and Stoica, I. Load Balancing in Structured P2P Systems. *Second International Workshop on Peer-to-Peer Systems* (2003).

[23] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A Scalable Content-Addressable Network . In *Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols* (San Diego, California, Aug. 2001).

[24] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (Nov. 2001), pp. 329–350.

[25] Singhal, S., and Cheriton, D. Using projection aggregations to support scalability in distributed simulation. In *Proceedings of the 16th International Conference on Distributed Computing Systems* (1996).

[26] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols* (2001).