

miChord: Decoupling Object Lookup from Placement in DHT-Based Overlays

Kathryn Chen, Loai Naamani, Karim Yehia
{kchen25, loai, kyehia}@mit.edu

ABSTRACT - Recently, Distributed Hash Tables (DHTs) have been a popular area of research. Existing DHTs make implicit or explicit assumptions about the homogeneity of a peer-to-peer system's participants and workload. In reality, peer-to-peer node and workload characteristics are not homogenous. As such, applications built on DHTs may opt to exploit this heterogeneity and prefer certain nodes over others. miChord uses pointers to decouple object lookup from object placement in Chord, enabling applications to influence the final location of object storage based on a preference of the application's choosing. This paper introduces the miChord decoupling scheme and discusses different metrics/properties of peers that can motivate such decoupling. We further provide simulation results showing performance gains over Chord when object placement is skewed in favor of different metrics.

I. INTRODUCTION

Over the past few years, the Internet has been witnessing the evolution of a new breed of innovative network architectures labeled as peer-to-peer networks. Peer-to-peer networks are characterized by direct access between peer computers rather than through a centralized server. Such systems constitute highly dynamic networks of peers with complex overlay topologies that are often unrelated to the physical network that connects the different nodes. Of the many features of recent peer-to-peer systems, efficient location of data items remains the core and most important operation [26].

Peer-to-peer systems can be differentiated by the degree to which they are structured: Is there a way of directly knowing which nodes contain which data items? Is a *random* search through the entire network necessary to find them? Structured networks, such as CAN [20], Chord [26], Pastry [22], Tapestry [31], and Viceroy [30], have recently emerged in an attempt to address the scalability issues caused by the inherently unscalable random search methods adopted by unstructured systems. In structured peer-to-peer systems, the overlay network topology is tightly controlled and objects are placed at precisely specified locations [2]. These systems use a distributed routing or hash table (DHT) to provide a mapping between the object identifier and location so that queries can be efficiently routed to the node where the desired data object is located.

With the advent of this class of structured systems, different research groups have explored a wide variety of applications, services, and infrastructures built on top of a DHT abstraction. Examples of such proposals include systems for wide-area distributed storage, indexing, search, querying [3], [8], web caching, application layer multicast, event notification [24], indirection services [29], and DoS attack prevention. As such, DHT systems hold great promise for the rapid deployment of Internet scale applications and services [28].

All of the aforementioned structured lookup algorithms make, in essence, some unrealistic explicit or implicit assumptions. These algorithms assume that all peers are uniform in resources [29] (such as storage capacity, network bandwidth, and CPU power) and that all peers are equally desirable to store prospective objects. However, measurements have shown that heterogeneity in deployed peer-to-peer systems is relatively extreme (with up to 3 orders of magnitude difference in bandwidth [1]). Therefore, the bottleneck caused by less capable peers could lead to the suboptimality of these existing lookup algorithms depending on the needs of the higher-level application. While these algorithms provide the basic service of data lookup, peer-to-peer *applications* provide high-level functionality using the underlying lookup algorithm. Much of the complexity in such systems arises because the environment of peer-to-peer systems is in reality very different from that of traditional distributed systems, such as those hosted by server farms. A single peer-to-peer system instance might simultaneously span many different types of constituent elements, such as dedicated servers, idle workstations, and old desktops [9].

We claim that it is possible to favorably separate object lookup from placement in DHT-based overlays using pointers. This separation would improve the performance of the overlying application according to the metric exploited. We choose to base our study on Chord. Performance is improved by publishing to nodes that have some relatively *desirable* property either to the publisher and/or to the overall underlying overlay. While this abstraction would only influence the final location of the actual object, a pointer to the node holding this object will be mapped to the node deemed responsible for it in conformance with Chord's regular consistent hashing process (see [26] for more details on Chord). Accordingly, locating the object is still deterministically bound and the

simplicity, provable correctness, and provable performance (lookup via $O(\log N)$ messages in steady-state) properties of Chord are all still preserved. Also, controlling data placement is in direct tension with the goal of a DHT, which is to uniformly distribute data across a system in an automated fashion [9]. Our level of indirection, however, is not meant to disrupt the generally desirable load balancing characteristics of Chord [26], but to only incrementally better align it with the publisher’s or system’s implicit desires through *constrained load balancing*.

In the context of a file system (CFS [7], PAST [22], Pond [20]) or file-sharing application (KaZaa, FreeNet [5], Gnutella, Napster), the aforementioned objects would be files and the sought-after resource would be storage. The mechanism to store published files would choose locations which are desirable to the publisher, based on some relative properties of the network and/or available nodes. Relevant properties of storage nodes might include up/downtime trends, bandwidth, number and size of files to store, duration of guaranteed storage, or latency-wise distance from the publisher. These and other properties will be maintained individually and communicated to fingering nodes when fingered as part of the periodic stabilization and finger table build-up and refreshing process defined by Chord. This would guarantee low metric state routing overhead and that properties of the nodes are as fresh as the finger table itself.

The rest of this paper is organized as follows. Section II provides background information on DHT-based object storage and lookup. Section III describes in detail the proposed object lookup-placement decoupling scheme using pointers and how it varies from basic Chord. In Section IV, we discuss various system characteristics and circumstances to be considered when selecting metrics that can be exploited by miChord. In Section V, we further discuss two sample metrics that miChord could exploit, namely 1) load balancing object serving requirements among peers in a miChord-enabled peer-to-peer system, and 2) optimizing overall object read latencies in the system. In Section VI, we introduce the miChord simulator and simulation environment before providing the methodology and results of our attempt to quantify the performance gain in optimizing for the two aforementioned metrics. We conclude in Section VII with discussing the tradeoffs incurred in divorcing object lookup from placement and summarizing the results of our findings and areas of future work.

A note on terminology used throughout the rest of this paper – The following words are used interchangeably: write, place, and publish, read, request, and download, data objects and files, and nodes and peers.

II. BACKGROUND

Several entirely distributed storage systems, such as CFS [7], PAST [22], and Pond [20], have been designed and implemented, however none are widely used. On the one hand, these systems demonstrate many desirable characteristics, including decentralized control, self organization and adaptation, sharing of system resources,

and scalability. On the other hand, in order for these systems to become prevalent, they have to perform efficiently, both with respect to read and write performance, and with respect to network resource consumption.

Most of the existing research on improving DHT performance has taken one of two approaches. The first approach is to minimize lookup latency by reducing the number of hops (one hop [9]). The idea is that if the average number of hops until a key lookup is resolved is reduced, then the average latency will be reduced as well. The second approach is to build the overlay so that it better reflects the underlying network [4]. The intuition here is that two nodes which are physically close in the underlying network should also have close virtual IDs so that when a lookup is performed, the actual packet does not have to cross the network unnecessarily many times.

In terms of storing pointers in DHTs, there are two systems that use this idea: Internet Indirection Infrastructure (i3) [27] and PAST [22]. i3 uses pointers as a decoupling mechanism in a DHT. However, the purpose of this decoupling in i3 is to offer a rendezvous-based communications abstraction rather than to enable a preference for data placement. PAST, a large-scale peer-to-peer persistent storage utility, uses pointers in *replica diversion*. To guarantee the durability of its data, PAST replicates each file k times. Files and nodes in PAST exist in the same Id space. A node with `nodeId` numerically closest to a file’s `fileId` is responsible for that file. Furthermore, when a node stores a file, the node creates replicas of that file at the $k-1$ nodes with `nodeIds` closest to the `fileId`. Replica diversion happens when a node A , one of the $k-1$ replica nodes, does not have enough storage capacity for the file. In such a case, A would ask another node B to store the file and A would store a pointer to B . miChord is not system-specific and is meant to enable a variety of object placement preferences.

A. CHORD

miChord consists of a layer on top of an enhanced Chord. Chord is fully described and evaluated in [26]. In this paper, we give a brief overview of Chord.

The Chord protocol uses a consistent hashing function such as SHA-1 to hash both the nodes’ IP addresses and files to produce node Ids and keys respectively. These identifiers are of length m -bits. The node identifiers are mapped onto corresponding positions in an identifier circle modulo 2^m . Each hashed key is then assigned to the node with the smallest identifier that is greater than or equal to the key’s identifier. This node is referred to as the key’s *successor* node. An example of this assignment of keys is shown in Figure 1(a), where the file with identifier “52” has been hashed to the node with identifier “55”. The consistent hashing function arranges the nodes on the Chord circle independent of the underlying topology. A number of load balance simulations have demonstrated that consistent hashing does not distribute keys evenly among all nodes, and that this variation increases linearly

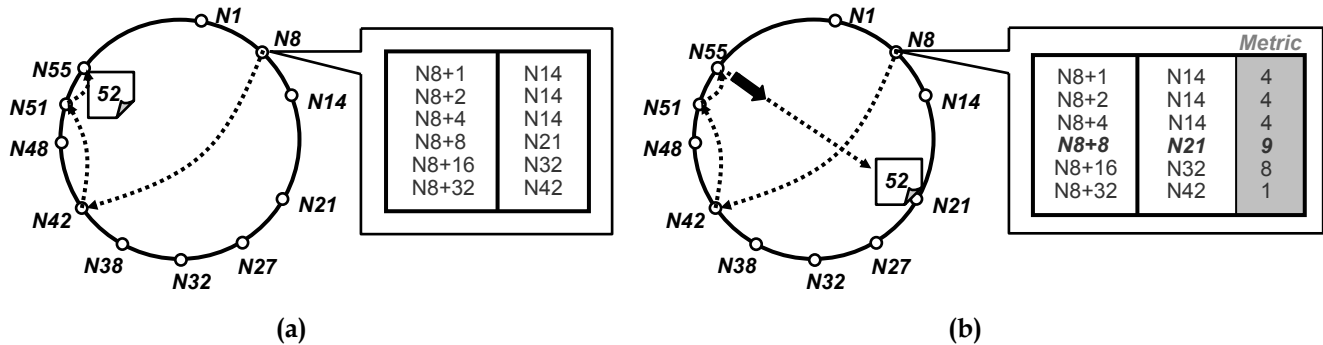


Figure 1. (a) Chord: A 6-bit identifier circle showing the nodes present in the system and node N8's finger table. The path corresponds to the query `lookup(52)` initiated by node N8 leading to object placement on N55. (b) miChord: The `lookup(52)` query initiated by node N8 now places the object on N21, which has the *optimum* metric in N8's finger table, and a *pointer* to the object on the query's result, N55.

with the number of keys in the system. One of the reasons for this imbalance is that nodes do not always cover the entire identifier space evenly [26].

Chord guarantees the correctness of lookups for objects in the system. A basic version of Chord maintains that each node need only know its successor in order to properly route queries; an enhanced version that resolves queries faster is one in which nodes store additional routing information in *finger tables*. Each i th entry in a node n 's finger table corresponds to the address of the first node that succeeds n 's identifier by at least 2^{i-1} . As a result, earlier entries in the table tend to be addresses of nearby nodes, whereas later entries correspond to nodes further on along the circle. This structure allows nodes to forward queries to other nodes in the remaining half of the identifier circle and cut down on forwarding time. Lookup is resolved with $O(\log N)$ forwardings with high probability [3]. This lookup mechanism is illustrated in Figure 1(a). Node N8 has been asked to lookup the document with hashed key "52". It looks in its finger table and forwards the query to the node having the greatest identifier that is less than or equal to "52", which in this case is the node N42. Thereafter, the lookup is refined by succeeding nodes.

In addition to specifying algorithms that maintain the network in the face of node joins and leaves, the Chord protocol implements a *stabilization* routine that keeps each node's successor pointers up-to-date. This ensures the correctness and speed of lookups when nodes join and leave the system, even if these dynamics occur concurrently. Stabilization is carried out periodically – it essentially verifies each node's current predecessor and successor, and refreshes the node finger tables.

In order to withstand the implications of node failure, each node keeps a "successor-list" of its r nearest successors. A node will use this list to replace its successor if it fails (i.e. leaves the network in an ungraceful manner). This is important because it ensures correct lookups before the network has stabilized, and equally important, it allows nodes to store replicas of the object in the r nodes to guarantee the existence of the object in the network at all times.

III. DESIGN

The miChord protocol seeks to place documents at locations based on some *preference*. It does so without compromising Chord's robust lookup mechanism. To influence placement of objects, a metric is associated with each node. Essentially, a node's metric is a quantification of how desirable the node is to store the object. Typically the metric is not constant and will evolve with the network dynamics. Sections IV and V present a further discussion on the subject of metrics.

In miChord, each node n on the identifier circle is aware of its own metric as well as the metrics of the nodes in n 's finger table. To support metrics, Chord finger tables have been enhanced to store metrics in addition to the identifiers and addresses of successor nodes. A node needs a mechanism whereby it "learns" the metrics of some other nodes. This learning is achieved as part of the stabilization routine. During stabilization, in addition to refreshing each node in the table's successor addresses, the routine also refreshes their metrics. Each node is responsible for maintaining and updating its own metric.

In addition to Chord enhancements, miChord consists of a layer on top of Chord called the *DataPlacer (DP)*. When a user wants to publish an object with Id k , the user tells the DP at some node n about the object (its Id) and the user's (or application's) publishing preferences. The DP then searches the set of nodes that n knows about through n 's finger table for the *preferred* node p that has the optimum metric in n 's known node set. Once p is found, DP sends the object to p for storage and places a pointer (pointing to p) at the actual successor node, s , of the object's Id, k . Thus, preference is granted based on a limited view of the world: whatever n sees of it through its finger table window. This is what we refer to as *constrained load balancing*.

When a request for k is made, the lookup query first yields k 's successor, s , in conformance with Chord, which will respond to the requestor with providing p 's address, now deemed responsible for k by miChord. An example of this mechanism is shown in Figure 1(b). Lookup first proceeds as it would with Chord. Then, upon reaching the successor node (node N55), a pointer directs the requester

to node N27, where the object has been placed by N8's DP. Also shown in Figure 1(b) is the modified miChord finger table that has an additional column to store node metrics.

IV. MOTIVATING METRICS

While DHT-based peer-to-peer overlays load-balance for keys and provide good key-lookup time guarantees, they do so at the expense of randomly controlling where data is placed. We believe the adverse side-effects of random data placement are very much application-specific. That is, depending on the higher-level application's requirements, peer-to-peer overlays can be performing suboptimally in different regards and different metrics can be exploited to alleviate this suboptimality. One clear disadvantage, for example, is that data may be undesirably placed on a node that is far from its owner/publisher or user. We see that the challenge of peer-to-peer computing goes beyond simple file-sharing [16], and hence we try to make our decoupling scheme metric-/property-agnostic and capable of propagating and promoting any node-specific property that is valuable to the overlying application and capable of being tracked, maintained, propagated, and chosen in accordance with the basic scheme described in Section III.

In [13] the authors assert that peer-to-peer applications form their dynamic user-level networks based on available bandwidth between peers and that overlay networks can configure their routing tables based on the bandwidth of overlay links. As much as available bandwidth is a sought-after resource in file-sharing applications and can be a metric miChord optimizes for, it is one that is relatively expensive to accurately and honestly track, mostly due to the intrusiveness of current available bandwidth measurement tools. Therefore the cost incurred in maintaining a given metric is a very important consideration that should be evaluated when using miChord to divorce information from location based on a specific metric.

Optimizing for bandwidth is not the only way to enhance performance. Although bandwidth in the core of the internet has been doubling at an incredible rate, latency has not been improving as quickly [13]. For small objects, latency, not bandwidth, is the dominating factor. If we consider distributed event notification [24], chatting, or gaming peer-to-peer systems, the sought-after resources could be significantly smaller and constantly interacted on by the publisher/users making them very latency-sensitive. Depending on the application, the original publisher may, for example, be the most important (if not only) benefactor miChord could serve. In a file-sharing application there are many others involved, but in a distributed file-system application, the publisher will be the *most* one interacting with his/her objects, so why not have them close? In a secure-backup system, the publisher will be the *only* one interacting with them. To serve the publisher best, one metric miChord could exploit is the distance between the publisher and the different nodes it is aware about in its finger table (or in that of a randomly contacted proxy node in the ring if the publisher is not a peer). The node yielding the smallest

distance, measured by RTT, with respect to the publisher could now serve as the best object storage option, based on the publisher's or proxy node's limited view of the world. In a collaborative file-system application, members of the workgroup (the *collaborators*) could be part of the same organization, department, or research group and are most probably within the same geographic location, hence making a storage node that is desirable (or close) to the publisher relatively desirable to the rest of the collocated collaborators. All the more, the object owner/publisher may choose to serve the object from his/her own node. In this case only a pointer to the publisher/owner is mapped to the node responsible for the object's key according to Chord.

Nodes can also easily keep track of their up/downtime or availability trends making highly available nodes better contenders for objects deemed critical by their publishers. Highly available nodes are also optimal locations for replicating data, as they tend to fail or leave the overlay less frequently causing less replication traffic in the system.

While peer-to-peer systems have been proposed as the solution to a diverse set of problems, many peer-to-peer systems will be used to present services to end users. End users are often skeptical of services that consume local resources in order to support anonymous outside users. User acceptance is often predicated on the extent to which end users feel they have fine-grained control over the intrusiveness of the service [2]. Accordingly, this metric-based abstraction can also be used to propagate nodes' different storage policies where relevant: How many files a node is willing to store? How big are the files permitted to be? How long is their storage guaranteed?

Ideally, miChord aims at supporting more than a single-metric-based decision (as in RON [1]) through cleverly taking into account all that a given node can say about itself (up/downtime, closeness to activity, available bandwidth, storage policy, lon./lat. coordinates, etc...). The node would then feed this information to some function either calculated by the publisher at the time of publishing (if it contains relative parameters, such as RTT between publisher and contending nodes) or readily calculated and propagated by the node itself when fingered. Again, this would be very dependent on the higher-level application and whether metric exploitation is built within the application logic or is at the subjective discretion of its users, namely object publishers.

V. SAMPLE METRICS

To demonstrate the usefulness of miChord and relevant considerations when picking a metric, we created two sample metrics. Each improves read performance in a different way. The first metric aims to load-balance the amount of data each node serves. The second metric aims to publish data to the *activity center* (to be defined later). In this section, we describe how the metrics are measured, calculated, and updated. In Section VI, we describe how we evaluated them and discuss the evaluation results.

A. Metric-1: Load Balancing the Amount of Data Served

The distribution of demand for real-data items is often skewed, leading to potentially poor load balancing, swamped nodes, and discarded requests [10]. Metric-1 aims to improve the overall read time by load-balancing the amount of data each node serves. The overall read time is the time from which a read is requested to the time at which all of the data is received. Improvement is most apparent in large files whose read time is bandwidth dominated. By load-balancing the amount of data each node is serving, we are optimizing the amount of bandwidth that is available to each request.

We made the following assumptions:

- Files have significantly different popularity, which is justified by real world observation [11].
- Writes are not concentrated at fewer than $\log_2 N$ nodes, which remain unchanged over time. This assumption is also justified by real world observation [25]. Since each node only has a partial view of the world, if writes are concentrated, there may be a set of nodes which have no data published to them because the publishing nodes do not know about them.
- Each node has the same uploading bandwidth. For nodes with different bandwidth capacities, multiple virtual instances can be instantiated on nodes with higher bandwidth to make this assumption appear to be true [7].

For this metric, each node keeps track of the amount of data D that it served in the last W seconds. W is the same for all nodes. Then the node's metric value is D . The best value for W depends on the frequency of stabilization. We want W to be small enough to reflect the current activity at a node, but big enough to compensate for the fact that the system will not react instantaneously to a change in a node's activity. As mentioned in Section III, a node's metric is propagated to some other nodes when stabilization occurs. Since a node's knowledge of other nodes' metrics is only as up-to-date as the last stabilization, the best value for W is directly related to the time between stabilizations.

B. Metric-2: Writing to the Activity Center

Metric-2 aims to improve the read performance by decreasing the overall latency in the system for read requests. Improvement is most apparent for small data objects whose read time is dominated by latency. The idea with metric-2 is that publishing data at or near the *center of activity* optimizes the read latency for the active users. We define *activity* to be the reads in the system. A node's metric-2 reflects how close it is to the activity center. The smaller its metric-2, the closer it is to the activity center. We measure distance using round trip times (RTTs) between nodes.

We made the following assumptions:

- Geographically speaking, activity is unevenly distributed. This characteristic is true in a system where all nodes are equally active, but unevenly distributed. For instance, some geographic areas are

denser with nodes than others, which is true in the real world.

- All nodes are equally likely to request the same object.

For this metric, each node's metric value is an exponential weighted average (EWMA). Upon receiving a read request from node j , node i updates i 's metric value M :

$$M_{\text{new}} = (k \times \text{RTT}_{i-j}) + (1 - k) \times M_{\text{old}}$$

A node's initial metric value M_0 is initialized to an estimate of the average latency between all nodes. M_0 should be the same for all nodes.

We expect that the propagation delay of the nodes' metrics will not be a significant performance penalty since we do not expect the location of activity to shift abruptly. Using the EWMA to keep track of the distance to the activity center, if the activity center shifts over time as happens in a global network with nodes in different time zones joining and leaving the system, each node's metric value will reflect the change accordingly.

There are limitations to this metric. By nature of trying to place object at or near the activity center, its performance is heavily dependent on the network of nodes and the activity pattern. Furthermore, placing all the data at a few central nodes may overwhelm those nodes. There are two factors that reduce this effect of overwhelming a few nodes. The first factor is that each node only knows about a subset of the nodes in the system. Thus, as long as writes are not concentrated at a few nodes, writes will be somewhat spread out. The second factor is that this metric is optimizing read performance for small data objects since those are the ones whose performance are dominated by latency.

Also, we would like a node's metric value to stabilize around a value that truly reflects a node's closeness to the activity center. When a node first joins, it is possible that one or two metric value updates with very long RTTs makes the node seem far from the center even though the node is actually close to the center. One way to prevent good nodes from being stuck with bad metric values is to introduce randomness. Use the metric for data placement 80% of the time (i.e 4 out of 5 times). The other 20% of the time, place the data randomly. A good node's metric value will recover when given the chance by randomness. Furthermore, randomness will also help in keeping data from concentrating at a few nodes.

VI. EVALUATION

A. Experimental Test Setup

The miChord simulator has been developed in C# (C-Sharp) over Microsoft's .NET framework and consists of 10 classes and around 1100 LOC. C# is a new type-safe and garbage collected language with a very rich class library for Rapid Application Development (RAD). The simulator

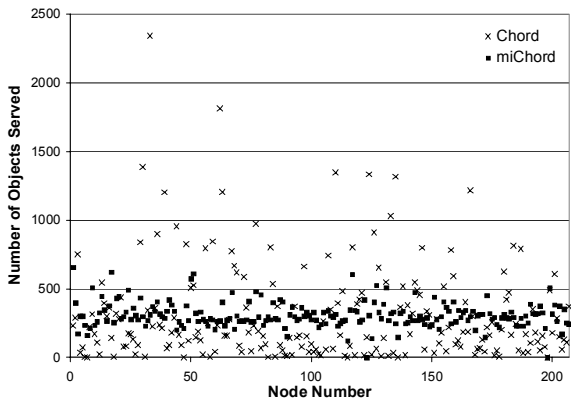


Figure 2. Scatter plot showing the number of objects served by Chord and miChord nodes in simulation 1.

simulates the `DataPlacer` layer described in Section III and the basic Chord lookup, node join, finger table generation, and stabilization operations specified in [26] and enhanced to support metrics. The simulator’s implementation included a `DataRequestor` layer, which upon honoring an object request, updates the relevant metric on the server node through: 1) incrementing the object serving counter of the node in the case of Section V-A, and 2) adjusting the RTT metric through an EWMA implementation as described in Section V-B. The up-to-date metrics are propagated to fingering nodes as they periodically update their finger tables.

Performance tracers are also inserted where relevant to keep track of the difference performance characteristics and results discussed later in this section. A separate `EventGenerator` module is responsible for generating the different events (node joins, file publishing, and file downloads) and their transitional delays. Events are probabilistically skewed to result in more file requests than writes, some nodes being more active than others, and some files being more popular than others. This makes our simulations better resembling of real-life peer-to-peer usage and traffic trends.

The generated event file is then fed simultaneously into 3 DHT-implementations: 1) basic Chord, 2) miChord attempting to load balance node’s object serving requirements (see Section V-A), and 3) miChord optimizing overall object read request latencies (see Section V-B). The output constitutes the performance results contrasted and discussed in depth below.

B. Simulation Environment

We use the PlanetLab testbed [6] as the underlying network topology in our simulations. The topology consists of the 207 production node list for which all-pairs-pings information is available. A matrix of the pair-wise latencies between all nodes (the average of which is around 150ms) is fed to the simulator and consulted whenever the RTT between two nodes is required to track performance during lookup and object publishing/retrieval. While the Chord Simulator developed in parallel with [26] keeps track of the total number of hops during a lookup operation, the miChord

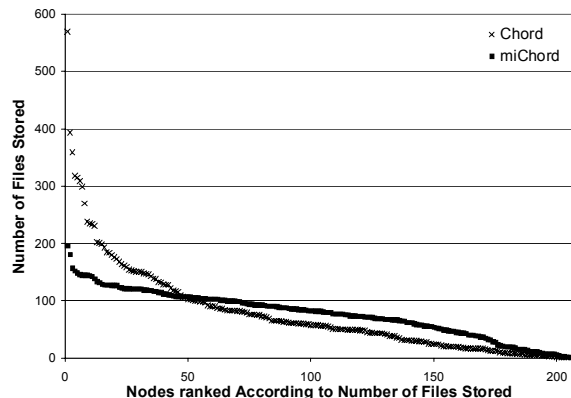


Figure 3. Scatter plot showing the number of files stored in Chord and miChord nodes at the end of simulation 1. Nodes have been ranked from largest number of files stored to smallest.

simulator also tracks the cumulative lookup latency incurred during lookup and object publishing/retrieval.

A total of 80,000 publish and read request events with varying in-between delays were generated. Of the 80,000 events, 16,000 were *publish* events, populating the simulation with 16,000 files of the same size with Ids in a 15-bit id space. The events generated reflect the assumptions stated in Section V. The results discussed hereunder are based on an EWMA constant k of value 0.9 (see Section V-B).

C. Experimental Results for Metric-1

Our simulations for metric-1 demonstrate that using metric-1 to place objects will result in better load balancing for the amount of data that each node serves. Numbering the nodes from 1 to 207, with the same node getting the same number in the Chord and miChord simulation, Figure 2 shows the number of read requests that each node served by the end of our simulation. As expected, miChord achieves much better load balancing when files have different popularities. The number of requests served by miChord are tightly clustered around 309, the number of requests each node would serve if the system load-balanced perfectly. The standard deviation for the number of files served is 383 and 93 for Chord and miChord respectively (75% improvement).

Metric-1 is only aimed to load-balance the nodes’ serving load, but we wanted to check how load-balanced the nodes’ storage was. Figure 3 ranks the nodes in each simulation in order from those storing the most files to those with the least files. The number of files each miChord node stores is more even, with a standard deviation of 40 compared to Chord’s 80 (50% improvement).

D. Experimental Results for Metric-2

As metric-2 is a more complicated metric than metric-1, we do a sanity check before proceeding to evaluate its performance. Once again numbering the nodes from 1 to 207, with the same node getting the same number in the

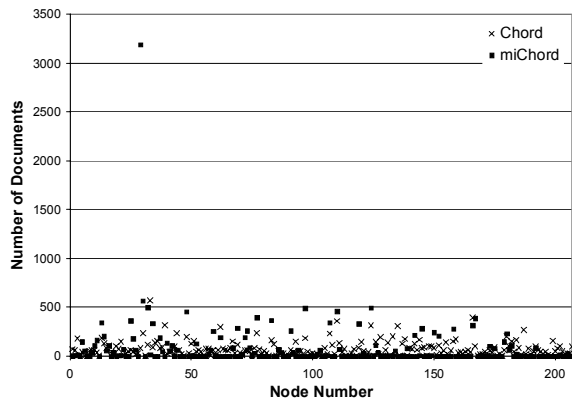


Figure 4. Scatter plot showing the number of files stored in Chord and miChord nodes at the end of simulation 2.

Chord and miChord simulation, Figure 4 shows how much data each node stored by the end of the simulation. We see a few miChord nodes that store many files (nodes near the activity center) while the rest store almost no files. Since metric-2 aims to minimize read latency, and object's whose read performance is dominated by latency tend to be small, we are less concerned about load-balancing the amount of data each node stores. However, by the end of our simulation results, the worst miChord node, where worst is defined by storing the most objects, stored 3,189 (or 18%) of the approximately 16,000 data objects. The worst Chord node stored 570 objects, one sixth that of the worst miChord node. Although ending up with data highly concentrated at a few nodes supports that metric-2 places data at or near the activity center, it also brings up scalability concerns. Before we propose possible solutions, let us discuss the performance gains.

Taking the average latency over all read requests, there was an 8% improvement for miChord requests over Chord requests. Our first observation is that 8% is small enough that it might be statistically insignificant. However, the large number of events in our simulation in combination with Figure 5, the cumulative distribution functions of latency in Chord vs miChord, and Figure 6, the average read latency experienced by each node in Chord vs miChord, leads us to

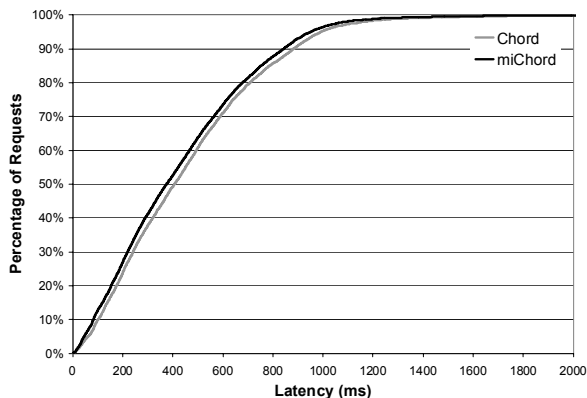


Figure 5. CDF chart showing the distribution of all Chord and miChord read request latencies in simulation 2.

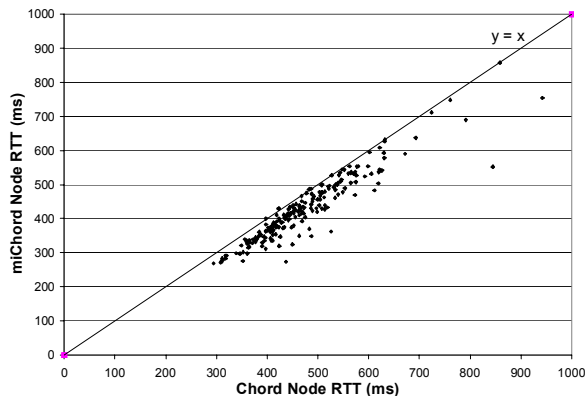


Figure 6. Scatter plot showing each node's average RTT in the Chord and miChord schemes by the end of simulation 2. Each point corresponds to a single node whose y-value is the average RTT in miChord and whose x-value is the average RTT in Chord. Points tend to be clustered below the $y = x$ line.

believe that there is in fact a small performance gain from placing data using metric-2. For Figure 6, each point below the $x=y$ line is a node request that had better average read latency in miChord; almost all points are below the $x=y$ line.

Moreover, the total request RTT in the above simulations is defined as the sum of the latency accumulated during the different hops required to resolve a certain lookup *and* the latency between the requestor and object server incurred when retrieving the object. miChord's intervention attempts to improve only this last retrieval hop between active requestors and servers. This does not directly affect the bulk of the total request RTT incurred during lookup; hence the marginal, yet consistent, 8% improvement.

As stated in Section V, the amount of improvement in read latency depends heavily upon topology. Should the topology merit the use of this metric, there are ways to work around data being too concentrated at a few nodes. One possible workaround idea is to place data according to two metrics. The first metric which reflects how much more data a node has than other nodes it knows about is used to determine whether a node is available for object placement. Of the available nodes, metric-2 is used to determine object placement. From another perspective, one might come to conclude that the need for optimizing a system by way of metric-2 can be easily alleviated by a simple caching scheme that guarantees a cached copy of popular objects in different active clusters. The interplay between caching, pointers, and replication is further discussed in the next final section.

VII. CONCLUSION AND FUTURE WORK

We have proposed the use of pointers to decouple object lookup from placement in DHT-based overlays. Since existing DHTs falsely assume homogeneity of peer-to-peer nodes and workloads, we claim and demonstrate that being able to influence the placement of an object in

favor of the overlying application's needs, and thus account for heterogeneity in the system, can lead to performance gains depending on the metric exploited.

One cost that comes with this scheme is that, in the face of a high churn rate, both the objects *and* the pointers need to be constantly replicated and maintained to ensure data availability and persistence. With pointers, there is now an additional point of failure during object retrieval besides its unavailability: the obsolescence/unavailability of the crucial pointer(s) that lead to it.

Furthermore, to preserve the preferences during initial object placement, a replication scheme more elaborate than the simple scheme of replicating to the r successor nodes (implemented in CFS [7] on top of Chord [26]) is required. This is because the r successors are unlikely to have the same desirable characteristics that made the original object holder the optimal placement choice. One possible workaround is to start with an r -successors replication scheme, and when the original holder fails/leaves the system, the secondary copy is migrated to a similarly *preferable* node and is elevated to primary copy status. Additionally, pointers to the primary copy need to be updated. This is an area of future work and a current limitation of the proposed scheme, especially if the initial placement preference need be maintained throughout the object's lifetime.

Just like any meaningful application using Chord is responsible for providing its desired caching and replication needs [26], applications layered on top of miChord are expected to complement and/or use the pointer scheme in implementing their caching and replication schemes. In fact, miChord *eases* the implementation of these features. Since pointers are relatively much cheaper (storage-wise) than the data objects they are pointing to, extensive caching of pointers can be made and less caching of the expensive objects - hence consuming less storage space and network traffic while guaranteeing more cache availability. Moreover, by replicating objects on nodes advertising *higher availability guarantees* (see Section IV), less replication traffic will be incurred since such nodes tend to fail/leave the system less often.

Finally, we would like to reiterate that miChord aims at eventually supporting more than a single-metric-based decision through cleverly taking into account a lot of what nodes can track and advertise about themselves. This information can be fed into some function either calculated by the publisher at the time of publishing (if it contains relative parameters, such as RTT between publisher and contending nodes) or readily calculated and propagated by the maintaining nodes when fingered. Again, this would be very dependent on the higher-level application's needs to control data placement. Accordingly, miChord is keen to remain generic and implementation-agnostic. It imposes as few constraints as possible on its adopting application, which can easily reduce it back to basic Chord by restoring data objects in place of their pointers.

VIII. ACKNOWLEDGEMENTS

We would like to thank Jeremy Stribling for kindly providing the PlanetLab latency data and Mike Walfish for his clarifications about the Chord Simulator. This work would have also not been possible without the continuous support and insight from Ben Leong and Prof. Hari Balakrishnan.

IX. REFERENCES

- [1] D. Andersen, H. Balakrishnan, F. Kaashoek, R. Morris, "Resilient Overlay Networks", In Proceedings of SOSP '01, October 2001.
- [2] S. Androutsellis-Theotokis, "A Survey of Peer-to-Peer File Sharing Technologies", available <http://citeseer.nj.nec.com/androutsellis-theoto02survey.html>
- [3] H. Balakrishnan, F. Kaashoek, D. Karger, R. Morris, I. Stoica, "Looking Up Data in P2P Systems", In Communications of the ACM, February 2003.
- [4] M. Castro, P. Druschel, et. al., "Exploiting Network Proximity in Distributed Hash Tables", In Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo), 2002.
- [5] I. Clarke, O. Sandberg, B. Wiley, T. W. Hong, "FreeNet: A distributed anonymous information storage and retrieval system", in "Designing Privacy Enhancing Technologies", Springer LNCS, New York, 2001.
- [6] B. Chun, D. Culler, T. Roscoe, A. Bavier et al., "PlanetLab: An Overlay Testbed for Broad-Coverage Services", In ACM SIGCOMM Computer Communication Review, January 2003.
- [7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS", In SOSP '01, October 2001.
- [8] D. Fensel, S. Staab, R. Studer, F. van Harmelen, "Peer-2-Peer Enabled Semantic Web for Knowledge Management", Wiley, London, UK, to appear.
- [9] L. Garcés-Erice, E.W. Biersack, P.A. Felber, K.W. Ross, G. Urvoy-Keller, "Hierarchical Peer-to-peer Systems", In Proceedings of ACM/IFIP International Conference on Parallel and Distributed Computing (Euro-Par), 2003.
- [10] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, P. Keleher, "Adaptive Replication in Peer-to-Peer Systems", available <http://citeseer.nj.nec.com/538119.html>
- [11] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, J. Zahorjan, "Measurement, Modeling, and Analysis of a Peer-to-Peer File Sharing Workload", ACM SOSP Proceedings, 2003.

- [12] N. Harvey, Michael B. Jones, S. Saroiu, M. Theimer, A. Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Properties", In Proceedings of USITS, March 2003.
- [13] K. Hildrum, J. Kubiawicz, S. Rao, B. Zhao, "Distributed Object Location in a Dynamic Network", In Proceedings of 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA), August 2002.
- [14] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski et al., "OceanStore: An Architecture for Global-Scale Persistent Storage", In Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.
- [15] B. Loblaw, H. Betchia, "Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables", available <http://citeseer.nj.nec.com/582367.html>
- [16] M. Oriol, "Peer Services: from Description to Invocation", In International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2002), 2002.
- [17] E. Pitoura, S. Abiteboul, D. Pfoser, G. Samaras, M. Vazirgiannis, "DBGlobe: A Service-Oriented P2P System for Global Computing", In ACM SIGMOD Record, September 2003.
- [18] R. Prasad, M. Murray, K. Claffy, C. Dovrolis, "Bandwidth Estimation: Metrics, Measurement Techniques, and Tools", IEEE Network, November 2003.
- [19] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems", In 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03).
- [20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, "A scalable content-addressable network", In Proceedings of ACM SIGCOMM, August 2001.
- [21] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiawicz. "Pond: the OceanStore Prototype." In Proceedings of Second USENIX Conf. File and Storage Technology, Mar. 2003.
- [22] A. Rowstron, P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems". In Proceedings of the 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms, November 2001
- [23] A. Rowstron, and P. Druschel "Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility ", In SOSp, 2001.
- [24] A. Rowstron, A. Kermarrec, M. Castro, P. Druschel, "SCRIBE: The Design of a Large-Scale Event Notification Infrastructure", In Networked Group Communication, 2001.
- [25] S. Saroiu, P. K. Gummadi, S. D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems", In Proceedings of Multimedia Computing and Networking 2002 (MMCN '02).
- [26] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications", In ACM SIGCOMM, August 2001.
- [27] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet Indirection Infrastructure", In Proceedings of ACM SIGCOMM, August 2002.
- [28] H. Zhang, A. Goel, R. Govindan, "Incrementally Improving the Lookup Latency of Distributed Hash Table Systems", In Proceedings of ACM SIGMETRICS, June 2003.
- [29] Y. Zhu, H. Wang and Y. Hu, "A Super-Peer Based Lookup in Highly Structured Peer-to-Peer Networks", In Proceedings of Parallel and Distributed Computing Systems (PDCS '03).
- [30] D. Malkhi, M. Naor, D. Ratajczak, "Viceroy: A Scalable and Dynamic Emulation of the Butterfly", In Proceedings of ACM Principles of Distributed Computing (PODC), July 2002.
- [31] B. Zhao, J. Kubiawicz, and A. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing", Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.