Structured Peer-to-Peer Overlays Need Application-Driven Benchmarks*

Sean Rhea†, Timothy Roscoe§, and John Kubiatowicz†
†University of California, Berkeley, §Intel Research Laboratory at Berkeley
srhea@cs.berkeley.edu, troscoe@intel-research.net, kubitron@cs.berkeley.edu

Abstract

Considerable research effort has recently been devoted to the design of structured peer-to-peer overlays*, a term we use to encompass Content-Addressable Networks (CANs), Distributed Hash Tables (DHTs), and Decentralized Object Location and Routing networks (DOLRs). These systems share the property that they consistently map a large space of identifiers to a set of nodes in a network, and while at first sight they provide very similar services, they nonetheless embody a wide variety of design alternatives. We present the case for developing application-driven benchmarks for such overlays, give a model of the services they provide applications, describe and present the results of two preliminary benchmarks, and discuss the implications of our tests for application writers. We are unaware of other empirical comparative work in this area.*

1 Introduction and Motivation

This paper reports on our ongoing work to devise useful benchmarks for implementations of structured peer-to-peer overlays, a term we use to encompass Content-Addressable Networks (CANs), Distributed Hash Tables (DHTs), and Decentralized Object Location and Routing networks (DOLRs). We argue that benchmarks are essential in understanding how overlays will behave in a particular application. Our work is driven partly by our experience implementing the OceanStore [6] and Mnemosyne [4] systems.

We want to benchmark structured peer-to-peer overlays for three reasons. The first is naturally for pure performance comparisons. However, in this paper we are not interested in declaring one overlay "better" or "worse" than another by measuring them on the same scale. The real value of application-driven benchmarks is to demonstrate how the design choices embodied in different overlay designs lead to different performance characteristics in different applications. Our aim is to relate three different areas: the design choices of the various overlays, their measured performance against our benchmarks, and the kind of performance and scaling behavior that users might see for their own applications. Our final motivation in benchmarking is our desire to provide overlay designers with a metric of success as expressed by application builders. Even a less-than-perfect benchmark would allow the designers of new algorithms to compare their work against previous designs, raising the barrier to entry for algorithms which hope to lure a large user base.

In the next section, we present a generic service model for structured peer-to-peer overlays which we use as a framework for measurement. Such a model attempts to capture the characteristics of an overlay which are of interest to an application writer. In Section 3, we describe our benchmarking environment, consisting of the Chord [13] and Tapestry [5, 16] implementations running on the PlanetLab testbed. In Section 4 we discuss two benchmarks for overlays, and the results of running them against our Chord and Tapestry deployments.

2 A Common Service Model

Before describing our benchmarking work, we present a generic service model for structured peer-to-peer overlays which we use as a framework for measurement. Such a model attempts to capture the characteristics of overlays which are of interest to an application writer.

The service model is in some ways like an Application Programming Interface (API), but it differs from an API in that it tries to capture the possible behavior of functionality presented to a user of the overlay, rather than explicitly specifying how the functionality is invoked. Furthermore, the model does not attempt to capture the routing mechanisms of the overlay except insofar as they manifest themselves in observed application performance, under "normal" conditions. We are not at this stage concerned with benchmarking overlays under attack in the ways described in [3], though this is clearly a direction for future work.

Figure 1 shows a functional decomposition of the services offered by various structured peer-to-peer overlays.[1] All existing overlays of which we are aware consist of an *identifier space,* $\mathcal{I}$ (often the space of 160-bit

* This research was supported by NSF Career Award #ANI-9985250, NSF Cooperative Agreement #ANI-0225660, and California MICRO Award #00-049.

[1] This figure is a simplified version of that presented in the work to establish a common API for such systems [2].

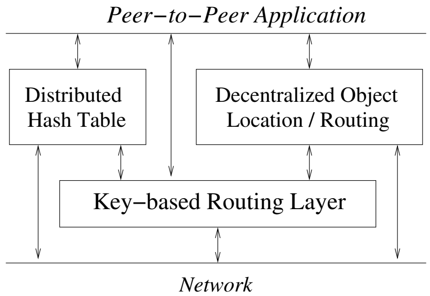

Figure 1: *Overlay Service Decomposition.* The functionality exposed by structured peer-to-peer overlays can be divided into a mapping of keys onto nodes, mechanisms to store and retrieve data items, and mechanisms to route to or locate data items stored according to a policy outside the overlay's control.

numbers), and a *node space,* $\mathcal{N}$ (often some subset of the set of IPv4 addresses), which consists of the nodes participating in the overlay at a given moment. The lowest level of an overlay, the *key-based routing layer* embodies a surjective mapping:

$$owner : \mathcal{I} \rightarrow \mathcal{N}$$

which maps every identifier $i \in \mathcal{I}$ to a node $n \in \mathcal{N}$. We chose the name $owner$ to embody the idea that the $owner$ is the node ultimately responsible for some portion of the identifier space. To compute $owner$, it is generally necessary to gather state from several successive nodes in the system; these nodes may all be contacted from the querying node itself, as in the MIT Chord implementation, or the query may be routed through the network, with each relevant node contacting the next, as in the Berkeley Tapestry implementation. In the original Chord paper [13], these two styles are respectively termed *iterative* and *recursive*; we will continue the use of this terminology in this work.

The most basic operations application writers are interested in is evaluating this function for some $i \in \mathcal{I}$, and/or sending a message to the node $owner(i)$. In Chord, the $owner$ function is directly implemented, and called $find\_successor$, while in Tapestry it is provided by the $route\_to\_root(i, m)$ function which sends a message m to $owner(i)$.

Above this basic facility many applications are also interested using overlays to store or retrieve data. To date, there are to our knowledge two different ways in which this functionality is achieved. In the first, a DHT is implemented atop the routing layer by mapping the names of data items into $\mathcal{I}$, and storing each object at the owner of its identifier.[2] In this case, an application may call the function $put(i, x)$ to store the datum x with name i, or $get(i)$ to retrieve the datum named i. For load balancing and fault tolerance, data items are often replicated a nodes other than the owner.

The second common method of implementing a storage layer in an overlay is to place data throughout the system independent of the overlay, but use the overlay to place pointers to where the data is stored. Algorithms using this second technique are called DOLRs to emphasize that they locate or route to data without specifying a storage policy. The functionality exposed by DOLRs consists of a $publish(i)$ operation, by which a node advertises that it is storing a datum with name i, and a $route\_to\_object(i, m)$ operation, by which a message m is routed to a node which has previously published i.

While many applications benefit from the higher levels of abstraction provided by some overlays, others are hindered by them. To Mnemosyne, for example, the additional performance cost of DHT or DOLR-like functionality is a disadvantage; in contrast, the DOLR properties of Tapestry are integral to the OceanStore design. We conclude this section by noting that the decomposition in Figure 1 is not a strict layering. While at a functional level, a DOLR may be implemented by a DHT and vice-versa, we show in the results section of this paper that there are performance consequences of doing so.

3 Experimental Setup

Our experiments are performed on PlanetLab [9], an open, shared testbed for developing and deploying wide-area network services. In our tests we use 83 of the nodes spread across the United States and Europe with up to three nodes at each site. While the hardware configuration of the machines varies slightly, most of the nodes are 1.2 GHz Pentium III CPUs with 1 GB of memory.

To gain some notion of the shape of the network, we performed two simple experiments. First, we pinged every host from every other hosts ten times and stored the minimum value seen for each pair; the results are graphed in Figure 2. The median inter-node ping time is 64.9 ms. Next, we had each node use the Unix *scp* command to transfer a 4 MB file from each of the machines planlab1.cs.caltech.edu, planetlab1.lcs.mit.edu, and ricepl-1.cs.rice.edu. This test is crude, but we only wanted a rough idea of the throughput available. The median observed throughput was 487 kB/s. The correlation between observed throughput and ping time is shown in Figure 3; as predicted analytically by Padhye et al. [8], TCP throughput and round-trip time show an inverse correlation. Since well-behaved peer-to-peer applications are likely to use TCP for data transfer in the foreseeable future, this correlation is significant; it implies that

[2] A common mapping of this sort is exemplified by CFS [1], which associates a data block b with an identifier $i = SHA1(b)$ and stores the block contents at node $n = owner(SHA1(b))$.

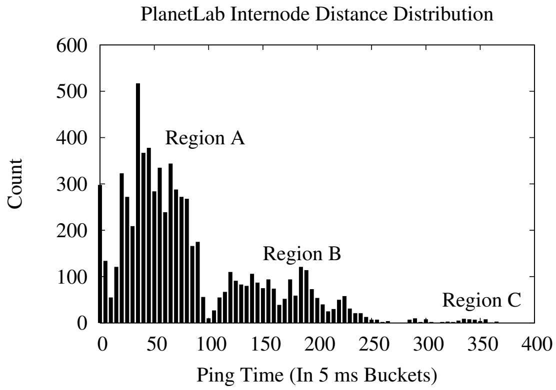

Figure 2: *Distribution of round-trip times in PlanetLab.* Region A (0–100 ms) contains 72.3% of the times, Region B (100–275 ms) contains 26.6%, and Region C (275–400 ms) contains the the negligible remainder.

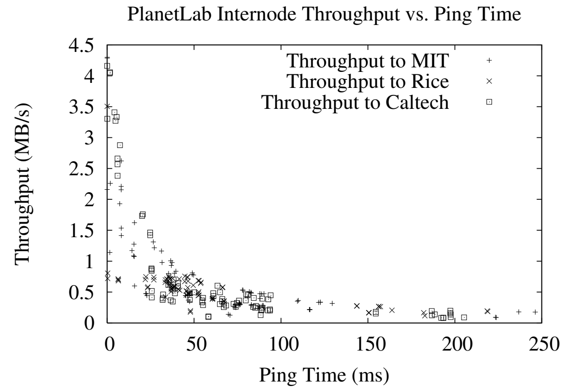

Figure 3: *PlanetLab Inter-node Throughput.* Throughput is inversely proportional to ping time.

nearby nodes are likely to observe higher throughput on data transfers than distant nodes.

Our experiments use the latest MIT implementation of Chord[3] as of September 23, 2002, and the Berkeley implementation of Tapestry[4] from the same date. Chord is implemented in C++ while Tapestry is implemented in Java atop SandStorm [15]. To test them both under the same framework, we extended the Chord implementation to export the *find_successor* functionality to the local machine through an RPC interface over UDP. We then built a stage which used this interface to provide access to Chord's functionality from within SandStorm. To test the overhead of this wrapping, we started a single node network and performed 1000 calls to *find_successor* from within SandStorm; the average call took 2.4 ms.[5] As we show below, this is a small percentage of the overall time taken by each *find_successor* operation.

To run an experiment with Chord, we start a Chord instance running this gateway for each node in the test, allow the network to stabilize, and then bring up the benchmarking code for each machine. To run an experiment with Tapestry, we bring up the Tapestry network first and allow it to stabilize, then we begin the benchmark.

4 Experimental Results

In this section we describe our experiments and analyze their results. While these experiments examine only a few performance characteristics of the designs, they demonstrate several counter-intuitive results of interest both to researchers in the area and application designers.

[3] Available at http://www.pdos.lcs.mit.edu/chord/.

[4] Available at http://oceanstore.cs.berkeley.edu/.

[5] This small delay only occurs once for each computation of the successor of a given identifier, not once per hop during that computation.

4.1 Find Owner Test

Our first benchmark measures the *find_owner* function. It tests the *find_successor* function under Chord; under Tapestry it is implemented as a *route_to_root* function with a response from the root to the query source over TCP. In the test, each node in the network chooses 400 identifiers uniformly and randomly from the identifier space and evaluates *find_owner* on each, timing the latency of each operation and waiting one second between them. All nodes perform the test concurrently.

Relevance: The *find_owner* functionality is used by almost every system built on an overlay. It is used to read and write data in CFS, PAST, and Mnemosyne, and to find candidate archival storage servers in OceanStore. With systems such as CFS and Mnemosyne, where the individual units of storage are small (on the order of disk blocks), the latency of the *find_owner* operation can drastically affect the time to perform a read or write.

Results: Figure 4 shows the median latency of the *find_owner* operation implemented using Chord as a function of the ping time between the query source and the discovered owner. Since in Chord the owner is not generally contacted when evaluating the function, the latency is roughly independent of the ping time. However, the distribution of nodes in PlanetLab causes the median latency to rise somewhat with increasing distance: if a query begins in a remote, sparsely populated section of the network, it will with high probability talk to the more populated portions of the network at some point. It is these same query sources that are likely to have a internode ping times in the upper portion of the range; hence the correlation.

Figure 5 shows the same graph, but using Tapestry to implement *find_owner*. In contrast to Chord, the owner is always contacted in computing this function in Tapestry,

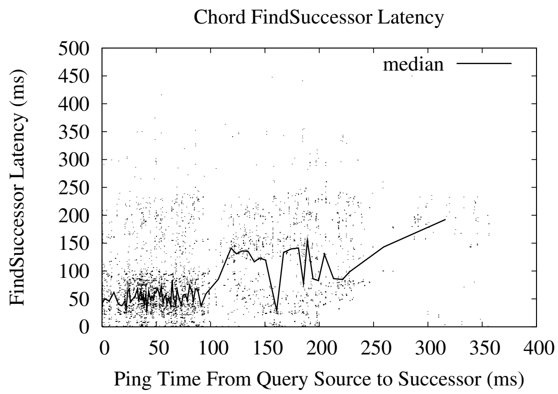

Figure 4: *Chord* find_owner *latency.*

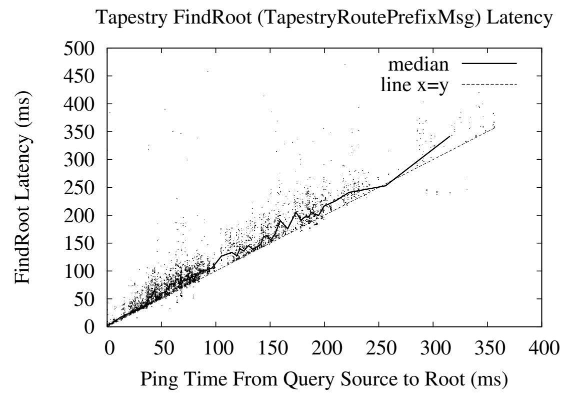

Figure 5: *Tapestry* find_owner *latency.*

Median latency of...	Chord	Tapestry
find_owner (measured)	62.3	85.2
call_owner (estimated)	94.8	51.1

Table 1: *Summary of* find_owner *results.* Times for the *call_owner* operation are estimated using the ping times shown in Figure 2. All times are in milliseconds.

so the *find_owner* latency should never be lower than the ping time. Ping times do vary between when they are measured and the test in question however, so some points fall below the line $y = x$. In general, though, Tapestry behaves as predicted in previous work [11]; the time to find the owner is roughly proportional to the network distance between the owner and the query source.

Discussion: A summary of the *find_owner* results is shown in Table 1. First, the median *find_owner* time in Chord is *less* than the median inter-node distance in PlanetLab. Given that each *find_owner* computation is expected to perform several RPCs, this result is somewhat surprising. We believe it arises from a combination of the skewed network distribution of PlanetLab and the optimization on Chord routing described in [1], whereby the latest Chord implementation chooses its next hop by weighting the distance traveled through the identifier space with the expected network cost of that hop; hops that travel furthest towards the destination identifier for the least cost are preferred. In our tests so far, there are still a fairly small number of nodes, resulting in only a few hops being necessary to compute owner. This means that a query starting on a node in Region A of Figure 2 would be likely to stay in within that region, where there are many short hops to choose.

Also shown in Table 1 are estimated times to send a message to the owner of an identifier using each algorithm; we call this operation *call_owner*. In Chord, it requires an extra network message to the owner, while in Tapestry it merely skips the return message used in the *find_owner* benchmark. Whereas Chord is 37% faster than Tapestry in our *find_owner* tests, we see that it is 86% slower on the estimated *call_owner* times. Given that the Chord implementation we used implemented iterative lookup, this result is not surprising. We expect an implementation using recursive lookup would be quicker.

We can also use the *find_owner* times to make a rough estimate of the median time to retrieve a data block from the owner. We first compute the median time to retrieve a block of negligible size. In Tapestry it is no different than the *find_owner* time, or 85.2 ms. In Chord, retrieving a block requires an extra round trip to the owner; adding the ping time to the owner to each of the points in Figure 4, we compute a median retrieval time of 121.9 ms.

Estimating the time to retrieve larger blocks is a matter of adding the time to transfer the additional bytes. Assuming the throughput between the query source and the owner is the median throughput of 487 kB/s, a 4 kB block takes only 8 ms, so a system like CFS could expect to see a 28% improvement in read time by using Tapestry. On the other hand, systems that fetch whole files from a single root (such as PAST) would not benefit as much. For example, for file sizes larger than 298 kB, the read time using Chord would be within 5% of the read time using Tapestry. There is much to be said for simplicity of design, and the more complex design of Tapestry is hard to justify without significant performance advantages.

Our computed times, of course, are only estimates; if such computations were sufficient for judging algorithmic performance, there would be no need for benchmarking. As such, we have already begun further testing to directly measure *call_owner* and block retrieval times.

4.2 Replica Location and Retrieval Test

Our next test is a replica retrieval test. In Chord, this test is simply the *get* operation implemented in DHASH,

Median latency of...	Chord	Tapestry
locate operation	60.5	64.7
ping to located replica	54.5	39.1
0-byte replica retrieval	116.8	64.7

Table 2: *Summary of locate benchmark results.* The replica retrieval times are estimated using the ping times shown in Figure 2. All times are in milliseconds.

the storage layer underlying CFS. In Tapestry, it is implemented as the *locate* operation, followed by a response over TCP with the requested data.

Relevance: An interesting feature of Tapestry is that it tries to route to the *closest* replica to the query source if more than one replica is available. In contrast, although DHASH generally provides several replicas of each data block in CFS, there is no direct mechanism in Chord to locate the nearest replica. Instead, it always locates the one stored on the *owner* of the block's identifier, and provisions to find closer replicas must be implemented at the application layer.

Locating nearby replicas has several benefits, the most obvious of which is performance; as Figure 3 shows, there is some correlation between ping time and throughput. Replicas close in ping time to the query source are more likely to have a high throughput path to the latter; they can thus not only deliver the first byte of a data item faster than replicas further from the query source, but they can also provide the last byte faster. Applications with high performance needs and multiple replicas of each data object should thus value locality highly.

In addition to performance, locality can also help provide availability; the closer a discovered replica is, the less likely it will fall on the other side of a network partition than the query source. Finally, by serving each read from the replica closest to the reader, a system may achieve better resource utilization and load balance.

Results: To test the locality features of the overlays, we first built a Chord network and stored 4 replicas of 10 different zero-byte objects as in DHASH. Then, we read the objects from each node in the system, one at a time, and recorded the total read latency and the node on which the replica was found. Next, we built a Tapestry network in which we stored the replicas on the same nodes as in the Chord test, and read the data as before. The time to retrieve a replica in Chord is computed at the time to discover a replica plus the ping time to that replica, minus the 2.4 ms wrapping overhead we described earlier.

After performing the tests, we calculated for each operation the distance to the closest replica from the query source, and the ping time to the replica that was actually discovered. A summary of the results is shown in Table 2. The first row shows the latency of the locate operation itself, while the second row shows the ping time from the query source to the replica which was located. While Chord finds a replica 7% faster than Tapestry, Tapestry finds replicas which are 39% closer to the query source. We can add a ping time to the Chord locate time as before to compute an estimated time to retrieve a zero-byte replica; with Tapestry the retrieve time is equal to the locate time. In this metric Tapestry is 81% faster than Chord. For reference, the median ping time to the closest available replica in each case was 28.8 ms, so Tapestry underperforms an ideal algorithm by 36%.

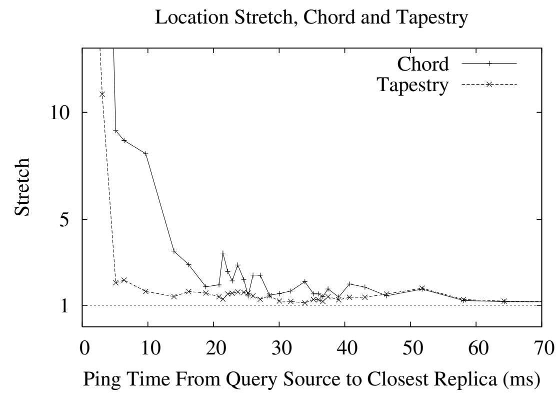

Figure 6: *Location stretch in Chord and Tapestry.*

Figure 6 graphs a value we term *location stretch*—the distance to the discovered replica divided by the distance to the closest available one. Location stretch is a measure of the *quality* of location; it shows the degree to which an overlay finds close replicas when they are available.

Discussion: We first observe that neither overlay performs well when the query source is within 5 ms of the closest replica; we believe that other techniques are necessary to achieve high performance in this range [11].

Next, we can see from Figure 6 that Tapestry significantly outperforms Chord when the closest available replica is within 5–15 ms of the query source. Referring back to Figure 3, a replica in this range generally sees high throughput to the query source as well, further increasing the benefits of locality.

We note finally that although Chord is not designed to find replicas according to locality, it could be extended to achieve low location stretch by finding all available replicas and then choosing the one with the lowest ping time. The CFS paper seems to imply that their implementation does something of this sort. Finding the latency to each replica would take time, but in some cases it might be justified. For example, if a *service* is being located (as in I^3 [14]), rather than a replica, if the number of replicas is very small, or if the replica is for a very large file, the location time may be dwarfed by the remainder of the operation. Further study is needed to determine the performance of such a scheme relative to Tapestry, and we plan to test the CFS implementation in our future work.

5 Conclusions and Future Work

One can observe structured peer-to-peer overlay designs from several angles: simplicity, robustness, efficiency, and/or performance. In this work we have focused on the latter, primarily because it is the easiest to measure, when the implementations of some of these algorithms are still in the early stages. Moreover, regardless of which of these features one studies, one can take an algorithmic approach, as in [10], or an application-level approach as we have taken. We view these two as complementary: at the same time that it is necessary for overlay builders to be exploring their design space, it is important for application writers to explore the differences between designs and the ways they affect the systems built on them.

In this paper we presented two benchmarks, *find_owner* and *locate*, evaluated over two overlays, Chord and Tapestry. We showed that for systems storing and retrieving blocks only from their *owner* nodes, *find_owner* provides insight into the choice of overlay. For systems that store entire files at the *owner*, there is little performance difference between Tapestry and Chord; for systems that store files as blocks, each on their own *owner* however, there is a small performance advantage to Tapestry. Depending on the application, however, this performance advantage may not be sufficient enough to justify the extra mechanism. Moreover, at least some of the latency advantage seen by Tapestry is due to its recursive—as opposed to iterative—routing style. In our future work, we plan to also study a recursive implementation of Chord so as to study each of these differences separately.

Our second benchmark, *locate*, showed that there is still work to be done on improving the ability of overlays to locate nearby replicas when they exist. We hope this result motivates the designers of these algorithms to further improve them; the correlation of throughput and ping times shown in Figure 3 indicate that there are significant performance gains available if they do.

Our ongoing work in the short term is to extend our benchmarks to other overlay implementations, and to track the increasing size of PlanetLab with more measurements. However, we believe we have only scratched the surface of the set of interesting and important benchmarks. We have not yet examined (for example) the cost of a new node joining a network, or the cost of one leaving. Neither have we examined the cost of a high rate of node turnover on a network, as highlighted by others [7, 12]. Finally, we have not analyzed the behavior of these overlays during node failure or maliciousness. The design of good application-driven benchmarks for such cases is a rich topic for future work. Nevertheless, we hope our existing work will help application designers to better understand the tradeoffs in choosing an overlay, and that it will motivate further design and implementation improvements by the networks' designers.

References

[1] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, 2001.

[2] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. of IPTPS*, 2003.

[3] M. Freedman, E. Sit, J. Cates, and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. of IPTPS*, 2002.

[4] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Proc. of IPTPS*, 2002.

[5] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed data location in a dynamic network. In *Proc. of ACM SPAA*, 2002.

[6] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[7] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Proc. of IPTPS*, 2002.

[8] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. of ACM SIGCOMM*, 1998.

[9] Larry Peterson, David Culler, Tom Anderson, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of HOTNETS*, 2002.

[10] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. In *Proc. of IPTPS*, 2002.

[11] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. of INFOCOM*, 2002.

[12] S. Saroiu, P. K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, 2002.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of SIGCOMM*, 2001.

[14] Ion Stoica, Dan Adkins, Sylvia Ratnasamy, Scott Shenker, Sonesh Surana, and Shelley Zhuang. Internet indirection infrastructure. In *Proc. of IPTPS*, 2002.

[15] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM SOSP*, 2001.

[16] B. Zhao, A. Joseph, and J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.