Kache : Peer-to-Peer Web Caching Using Kelips

PRAKASH LINGA, INDRANIL GUPTA and KEN BIRMAN
Cornell University

To achieve high performance, the emerging generation of Web-based database and Web Services systems will need to take full advantage of caching; latencies associated with their three or four-tier architectures would otherwise be prohibitive. Cooperative caching supported by peer-to-peer (p2p) indexing has been suggested as a scalable way to gain these benefits, but many performance concerns have not yet been addressed in this arena. Lookup latencies are required to be low, and the overhead and the potential for disruption as nodes join and leave the system need to be minimized. The second consideration is important because this kind of "churn" is known to disrupt many p2p technologies. Our paper investigates the issue experimentally. We describe a cooperative caching system called Kache*, which we implemented over a p2p index called* Kelips*, and evaluate it in a trace-driven experiment during which failures and other kinds of disruptions were injected. Under quiescent conditions, Kache can perform a lookup in* one hop: *a node seeking information can find it with high probability by querying just one other node that is topologically near-by, irrespective of system size. Even in settings subject to significant churn, Kache rapidly restabilizes after disruption, and the same cache hit rates seen in the quiescent case can be maintained at small additional cost. We conclude that Kache could be a good choice in settings where developers seek to reduce load on a shared server, or where a cluster of clients can communicate among themselves cheaply but incur long delays when communicating to a server.*

Categories and Subject Descriptors: H.3.7 [**Information storage and retrieval**]: Information Search and Retrieval - Search process; H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing - Indexing methods

General Terms: Design, Performance

Additional Key Words and Phrases: Web Caching, DHT, churn, fault tolerance, scalability, locality, load balancing

1. INTRODUCTION

Caching is receiving new scrutiny in light of the growing popularity of the *Web Services architecture,* in which a client may be separated from the servers it uses by two or three tiers of intermediaries (even more when message queuing mechanisms are employed to increase availability). An *external caching* scheme consists of a store in which applications can place copies of web objects and a lookup mechanism whereby requests can be satisfied from the cache. An external cache is *cooperative* if stores associated with different clients share their contents, so that one client's request can be satisfied from a different client's cache. The focus of this paper is

on technology support for cooperative external caching.

Web services deployments may be very large, hence there is a need for loosely coupled solutions that impose minimal overhead and require little management. To succeed, a technology must scale well, be self organizing and self repairing, and tolerate the transient disruptions common in large networks. Unfortunately, the distributed systems community has discovered that *churn*, caused by brief network outages or by nodes joining and leaving can cripple the most common file sharing technologies, which would otherwise be obvious candidates for cooperative caching.

This paper presents `Kache`, a new peer-to-peer web caching system [Linga et al. 2003]. Kache is designed by using *Kelips*, a peer to peer indexing protocol [Gupta et al. 2003]. Kache uses probabilistic techniques to heal proactively in the face of system instabilities. The system also adapts itself to the underlying network topology to provide access to nearby cached copies of web objects. Our study focuses on a conventional web caching scenario, in part to facilitate implicit comparison with prior work [Iyer et al. 2002; Padmanabhan and Sripanidkulchai 2002; Wang et al. 2002], and in part to take advantage of the high-quality trace data available for this case. However, we believe that Kache would be equally applicable in other kinds of Web services or distributed database settings, and that the Kelips protocol underlying Kache could also be used in other kinds of indexing applications.

Mohan surveys a variety of these scenarios in [Mohan 2002]. For example, dynamically generated web content delivery can be accelerated by caching html fragments, web applications such as enterprise java beans (EJBs) can be cached, and database operations such as queries can be speeded up by caching their results. Peer-to-peer indexing is a good match for such settings, and the longer term goal arising from this paper is to develop a powerful caching solution useful in a diversity of web caching settings.

A primary concern about peer-to-peer cooperative caching is that while protocols in this class scale well, they are also sensitive to "churn", whereby hosts that join and leave the system trigger high overheads as the system restabilizes [Padmanabhan and Sripanidkulchai 2002]. Churn-related overhead is more than just a nuisance, since an attacker seeking to disable a system could provoke churn to mount a distributed denial of service attack, potentially crippling the sharing mechanism while also subjecting participating machines to high loads. Resistance to churn is a crucial objective if this type of mechanism is to be successful.

The Kelips[1] system is unusual in employing probabilistic schemes and a self-regenerating data structure that adapts automatically and with bounded loads (independent of system size) as machines join, leave, or fail, or other disturbances occur. Here we show that when Kelips is employed for shared web caching, the system maintains rapid lookups and low overheads even when subjected to high churn rates, and even if new cache entries are simultaneously added.

Our analysis starts by exploring the overall performance of Kache under a range of normal conditions. The analysis includes server-bandwidth, lookup time and access latency both when a cache hit occurs and when a miss is detected, and

[1]The name of our system was derived from *kelip-kelip*, a Malay name for the self-synchronizing fireflies that accumulate after dusk on branches of mangrove trees in Selangor, Malaysia [Website].

quality of load balancing. We then explore robustness of the system to failures and churn.

Our work is experimental, and is based on a prototype implementation. We undertook a microbenchmark study, by running Kache within a small cluster, and a trace-drive simulation study for larger-sized systems. Even the simulation uses the real code; we simply link it to a library that permits us to run the code against a synthetic workload. The evaluation uses web access traces from the Berkeley Home IP network [Davison], transit-stub network topology maps obtained through the Georgia-Tech generator [GTech], and churn traces from the Overnet deployment (obtained from the authors of [Bhagwan et al. 2003]).

We show that loads are low and independent of the rate of churn and failure events, and the system adapts itself so that peers that tend to join and leave frequently are unlikely to be used as targets in lookups - in effect, requests are directed towards more reliable peers and, within this set, towards those with lower expected latency. Coupled with the extremely good scalability of the technique, we believe that Kelips is a strong candidate for caching in web systems of all kinds, including traditional web sites, databases, and web services.

Paper Organization. Since the Kache system is built using Kelips, the organization of the paper is as follows. Section 2 describes the Kelips system, then Section 3 presents the design of Kache. Section 4 presents experimental results for both Kelips and Kache. We present related work in Section 5 and conclude in Section 6.

2. KELIPS

2.1 Peer-to-peer indexing structures

A common use for peer-to-peer (p2p) protocols is to implement an index over a set of participating processes (nodes), whereby applications can insert (*key,value*) pairs into an indexing structure, and can perform lookup operations on keys, retrieving (w.h.p.) the associated value.[2] Such systems are sometimes called *distributed hash tables* or *DHT*s. The name is appropriate because objects can be inserted, retrieved, and deleted from a distributed collection of nodes; this is analogous to "buckets" in the classical hash table.

For cooperative web caching, the *key* of a cached web object copy would consist of its original URL, and the *value* would specify the location of the cached copy. For simplicity, the discussion of Kelips in this section implicitly assumes that each inserted resource has a unique key, i.e., each web object has at most one cached copy anywhere in the system. When we discuss Kache in Section 3, we will modify the Kelips design to take advantage of multiple cached copies of the given web object.

Such peer to peer DHTs are intended for large-scale deployments, and include functionality whereby nodes can join or leave the system; failed nodes are automatically detected and then excluded. Well known examples include Chord [Dabek et al. 2001], Pastry [Rowstron and Druschel 2001], and Tapestry [Zhao et al. 2001].

[2]We note that whereas many DHT systems treat data replication as well as lookup, our work focuses only on the lookup problem, leaving replication to the application. For reasons of brevity, this paper also omits any discussion of privacy and security considerations.

DHT architectures reflect design tradeoffs between the amount of storage overhead at each node, the communication costs incurred while running, and the costs of resource retrieval. The works cited above adopt design points in which storage costs are logarithmic in system size and hence small, and lookup costs are also logarithmic (unless cache hits shortcut the search). It is not clear that this is a good finding: a lookup that must visit a logarithmic number of nodes to find the data of interest could be very slow, since each hop involves sending a message to a machine that may be very remote within the Internet, and some of those machines will probably have slow connections, be heavily loaded, or have departed from the system unnoticed by the nodes that point to them. Thus even a fairly short path can incur extremely long delays. This observation motivated us to look at other design points.

Kelips is a DHT of our own design[Gupta et al. 2003] in which we increase the soft state memory usage and accept a steady background network communication overhead to force lookup costs down to a single hop. That is, a process performing a lookup needs to ask just a single node to find the information it seeks. This guarantee is probabilistic: Kelips gains scalability in part by accepting a somewhat relaxed consistency goal, i.e., a node may be missing a small percentage of (key,value) tuples that it is supposed to be storing. But because the probability of errors is low and each node is independent, one can easily compensate by concurrently querying several nodes, driving the probability that the desired (key,value) mapping won't be found down exponentially quickly as a function of the number of nodes queried.

A zero-hop lookup can be achieved by simply replicating the full state of the DHT at all nodes. However, such an approach would scale poorly and, because the replication method used is probabilistic, in many cases additional one-hop queries would still be needed to achieve a high quality result. Kelips settles on a design point in which all queries can be answered in one hop, and $O(\sqrt{n})$ space is employed at each node, where n is the number of nodes in the DHT. The $\sqrt{n}$ design point is of interest because it strikes a balance between the storage required by each node to keep track of other members of the system; a Kelips node maintains a list of peers of size $O(\sqrt{n})$. The storage required for soft state; a Kelips node maintains replicas of the soft state associated with $O(\sqrt{n})$ other nodes.

The probabilistic consistency of Kelips manifests itself in the following way: there is a small probability that a correctly functioning node may nonetheless lack membership information that it "should" be tracking, or lack replicas of soft state that "should" have mapped to it. In fact, all DHTs can be inconsistent: at a given instant in time, a system like Chord, Pastry or Tapestry may have a considerable number of incorrect finger-table entries, or may lack data that should be mapped onto it. This occurs because these systems all need a quiescent period after a node joins, leaves or fails, or a new (key,value) pair is inserted. But in distinction to these other systems, Kelips bounds the degree of inconsistency that can arise, and explicitly *embraces* the probabilistic nature of replication in its design, using explicitly randomized protocols to replicate new (key,value) pairs, and to select the peer to query when a lookup occurs. This randomized mechanism is biased using an idea from the "small worlds" algorithms to increase the likelihood that a query

will be satisfied by a nearby node without simultaneously increasing the likelihood that a query will fail to find the desired (key,value) pair.

Kache uses Kelips to maintain cooperative cache indices and to perform lookups. Even with its $\sqrt{n}$ design point, memory usage is small for systems with moderate sizes - if 10 million objects are inserted into a 100,000-node system, Kache uses only 1.93 MB of memory at each node. The system exhibits stability in the face of node failures and packet losses, and hence would be expected to ride out "churn" arising in wide-area settings as well as rapid arrival and failure of nodes. This resilience arises from the use of a lightweight epidemic multicast protocol for replication of system membership data and resource indexing data [Bailey 1975; Demers et al. 1987].

2.2 Core Design

Kelips consists of k virtual *affinity groups*, numbered 0 through $(k-1)$, where k approximates $sqrt(n)$ and is known to all nodes in the system. Each node lies in an affinity group determined by using a consistent hashing function to map the node's *identifier* (IP address and port number) into the integer interval $[0, k-1]$. Let n be the number of nodes currently in the system. The use of a cryptographic hash function such as SHA-1 ensures that with high probability, each affinity group contains close to $\frac{n}{k}$ nodes.

The *soft state* of a node consists of the following entries:

• Affinity Group View: A list of other nodes lying in the same affinity group. Each entry carries additional fields such as round-trip time estimate, heartbeat count, etc. for the other node. The view need not be consistent, but we do assume that the union of all lists covers the full set of nodes with identifiers that map to the group.

• Contacts: For each of the *other* affinity groups in the system, a small (constant-sized) set of nodes lying in the foreign affinity group. Entries contain the same additional fields as in the affinity group view.

• Resource Tuples: A (partial) set of tuples, each detailing a resource name and host IP address of the node storing the resource or object; this node storing a resource is called the resource's *homenode*. A node stores a resource tuple only if the resource's name hashes to this node's affinity group. Resource tuples are also associated with heartbeat counts.

Figure 1 illustrates an example. Entries are stored in AVL trees to support efficient operations.

Memory Usage at a node The total storage requirements for a Kelips node are $S(k,n) = \frac{n}{k} + c \times (k-1) + \frac{F}{k}$ entries (c is the number of contacts per foreign affinity group and F the total number of resources present in the system). For fixed n, $S(k,n)$ is minimized at $k = \sqrt{\frac{n+F}{c}}$. Assuming the total number of resources is proportional to n, and that c is fixed, k then varies as $O(\sqrt{n})$. The minimum $S(k,n)$ varies as $O(\sqrt{n})$. This is larger than Chord or Pastry, but reasonable for most medium-sized p2p systems.

Consider a medium-sized system of $n = 100,000$ nodes over $k = \lceil \sqrt{n} \rceil = 317$ affinity groups. Our current implementation uses 60 B resource tuple entries and 40 B membership entries, and maintains 2 contacts per foreign affinity group. Inserting

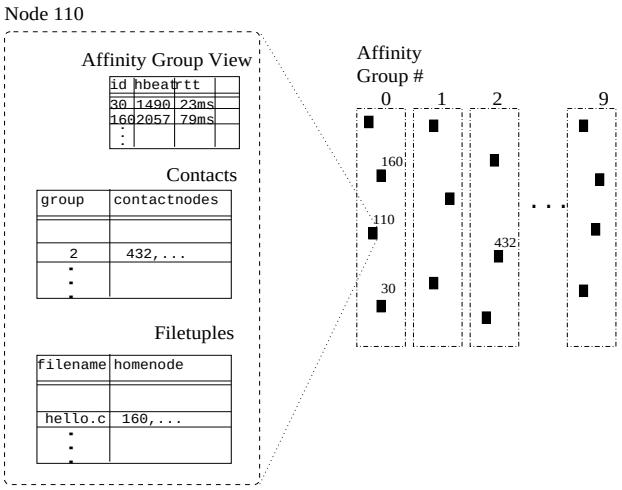

Fig. 1. **Soft State at a Node:** *A Kelips system with nodes distributed across 10 affinity groups, and soft state at a hypothetical node.*

a total of 10 million resources into the system thus entails 1.93 MB of node soft state. With such memory requirements, resource lookup queries return the location of the resource within $O(1)$ time and message complexity (i.e., these costs are invariant with system size n).

2.2.1 *Background Overhead.* Given a system of n nodes across k affinity groups, view, contact and resource tuple entries are refreshed periodically within and across groups. This occurs through a heartbeating mechanism. Each view, contact or resource tuple entry stored at a node is associated with an integer heartbeat count. If the heartbeat count for an entry is not updated over a pre-specified time-out period, the entry is deleted. Heartbeat updates originate at the responsible node (for resource tuples, this is the homenode of the resource, i.e., the node storing the resource) and are disseminated through a peer-to-peer Epidemic protocol [van Renesse et al. 1998].

We briefly describe epidemic-style dissemination within an affinity group. Then we generalize to multiple affinity groups.

An epidemic (or gossip-based) protocol disseminates a piece of information (e.g., a heartbeat update for a resource tuple) in the following manner. Once a node receives the piece of information to be multicast (either from some other node or from the application), the node gossips about this information for a number of *rounds*, where a round is a fixed local time interval at the node. During each round, the node selects a small constant-sized set of target nodes from the group membership, and sends each of these nodes a copy of the information[3]; these nodes now become infected and begin to gossip about the information, and so forth. Once

[3]Our system actually uses what is called *push-pull* gossip: in addition to forwarding information to a target node, if that target node is infected with information lacking on the originating node, the target can send back a copy of its own information, thus infecting the originator.

infected, a node cannot become reinfected by the same information.

It should be easy to see that, with high probability, the protocol transmits the multicast to all nodes. The latency varies with the logarithm of affinity group size. Gossip messages are transmitted via a lightweight unreliable protocol such as UDP; the protocol does not need to guarantee reliability. Gossip target nodes are selected through a weighted scheme based on round-trip time estimates, preferring nodes that are topologically closer in the network. Kelips uses the spatially weighted gossip proposed in [Kempe et al. 2001] towards this. A node with round-trip time estimate rtt is selected as gossip target with probability proportional to $\frac{1}{rtt^r}$. As suggested in [Kempe et al. 2001], we use a value of $r = 2$, where the latency is polylogarithmic ($O(log^2(n))$.

Analysis and experimental studies have revealed that epidemic style dissemination protocols are robust to network packet losses, as well as to transient and permanent node failures. They maintain stable multicast throughput to the affinity group even in the presence of such failures. See references [Bailey 1975; Birman et al. 1999; Demers et al. 1987].

Information such as heartbeats also need to propagate across affinity groups (e.g., to keep contact entries for this affinity group from expiring). This is achieved by selecting a few of the contacts as gossip targets in each gossip round. Such cross-group dissemination implies a two-level gossiping scheme [van Renesse et al. 1998]. With a uniform selection of cross-group gossip targets, latency is more than that of single group gossip by a multiplicative factor of $O(log(k))$ (same as $O(log(n))$).

Gossip messages in Kelips can carry many resource tuples and membership entries. This includes entries that are new, were recently deleted, or with an updated heartbeat. Since Kelips limits bandwidth use at each node, not all the soft state can be packed into a gossip message. Maximum rations are imposed on each of the number of view entries, contact entries and resource tuple entries that a gossip message may contain. For each entry type, the ration subdivides equally for fresh entries (ones that have so far been included in fewer than a threshold number of gossip messages sent out from this node) and for older entries. Entries are chosen uniformly at random, and unused rations (e.g., from few fresh entries) are filled with older entries.

Ration sizes do not vary with n. With $k = \sqrt{n}$, this increases dissemination latencies a factor of $O(\sqrt{n})$ above that of the Epidemic protocol (since soft state is $O(\sqrt{n})$). Heartbeat timeouts thus need to vary as $O(\sqrt{n} \times log^2(n))$ for view and resource tuple entries, and $O(\sqrt{n} \times log^3(n))$ for contact entries.

These numbers thus are the convergence times for the system after membership changes. These compare favorably with convergence times for other existing peer to peer DHTs, e.g., [Dabek et al. 2001], where the convergence times often grow super-linearly as a function of increase in system size. Kelips' convergence times are achieved through only the gossip messages sent and received at a node (henceforth called the *gossip stream*). This imposes a constant per-node background overhead. The gossip stream keeps heartbeats flowing in spite of node and packet delivery failures, thus allowing lookups to succeed.

2.2.2 *Resource Lookup and Insertion.* **Lookup:** Consider a node (querying node) that desires to fetch a given resource. The querying node maps the resource name

to the appropriate affinity group by using the same consistent hashing used to decide node affinity groups. It then sends a lookup request to one or more topologically close contacts among those known for that affinity group. A lookup request is resolved by searching among the resource tuples maintained at the node, and returning to the querying node the address of the homenode storing the resource. This scheme returns the homenode address to a querying node in $O(1)$ time and with $O(1)$ message complexity. The querying node can now obtain the resource itself from its homenode. The number of concurrent query requests will be a small constant and can be varied to compensate for the risk that the target system might lack the desired data; we discuss this in more detail shortly.

Insertion: A homenode h that wants to insert a given resource f, maps the resource name to the appropriate affinity group, and sends an insert request to the topologically closest known contact for that affinity group. A new resource tuple is created listing the resource f as being stored at h, and is inserted into the gossip stream. The information spreads to all nodes in the affinity group in $O(log(\sqrt{n}))$ time, and since gossip occurs at a constant rate and bounded message size, will do so with $O(1)$ message complexity.[4] The homenode periodically refreshes the resource tuple entry to keep it from expiring.

As was noted in the introduction, factors such as inaccurate contact sets or incomplete resource tuple replication might cause a one-hop lookup or insertion to fail. Biased partial membership information might cause uneven load balancing. These problems are addressed by the concurrent query scheme we present in Section 2.3.

2.3 Auxiliary Protocols and Algorithms

Next, we outline the protocol used in Kelips to handle node arrival, membership and contact maintenance, topological considerations and multi-hop query routing.

Joining protocol: Like in several existing p2p systems, a node joins the Kelips system by contacting a well-known introducer node (or group). For example, a well-known http URL could be used to contact a database that keeps a partial list of existing members. Once an introducer is located, it provides the joining node with a system view that it uses to warm up its soft state and start gossiping and populating its view, contact and resource tuple set. News about the new node spreads quickly through the system.

Node Failure and Deletion: A node that has failed or removed itself from the Kelips system will stop relaying heartbeats for itself as well as the files stored at it (i.e., for which it is the homenode). This will cause such entries to expire and be deleted at all other nodes in the system. Failure detection can be speeded up by explicitly propagating, through the gossip stream, identifiers of nodes that are suspected to have failed.

Spatial Considerations: Each node periodically pings a small set of other nodes it knows about. Response times are included in round-trip time estimates used in

[4]This assumes that the rate of updates is lower than the "capacity" of the system to propagate them. If a temporary period of rapid updates occurs, the limit on the size of gossip messages would cause the epidemic to run for a longer period of time; indeed, given a sustained period of rapid updates, affinity group member could fall far behind. Our work doesn't address this issue, because such scenarios seem unlikely in the settings for which Kache is designed.

spatial gossip.

Contact maintenance: The maximum number of contacts is fixed, yet the gossip stream constantly supplies potential contacts. *Contact replacement* policy can affect lookup/insert performance and system partitionability, and could be either proactive or reactive. Currently, we use a proactive policy with the farthest contact chosen as victim for replacement.

Multi-hop Query routing: We have noted that there is some probability that a query could fail even when a resource is actually present in the system, because of the various factors cited earlier. Because this is an unlikely event, and we wish to minimize the message overhead of our system, Kelips starts with a single query in the hope that it will locate the desired resource rapidly. However, if a resource lookup or insert query fails, the querying node retries the query in a more aggressive *multi-hop* mode. Query retries occur along several axes: a) the querying node can concurrently send the query to multiple contacts, b) contacts could be asked to forward the query within their affinity group (up to a specified TTL), c) the querying node could request the query to be executed at another node in its own affinity group (if this is different from the resource's affinity group). Notice that any successful reply will suffice. Query routing occurs as a random walk within the resource affinity group in (b), and within the querying node's affinity group in (c). TTL values on multi-hop routed queries and the maximum numbers of tries define a tradeoff between lookup query success rate and maximum processing time. The normal case lookup processing time and message complexity are unchanged by this extension, and since the need for multi-hop queries is very rare, the mechanism imposes little overhead in the runs we've studied.

Resource insertion occurs through a similar multi-hop multi-try scheme.

2.4 Kelips Flexibility

While employing Kelips in a p2p application (such as web caching), the designer as well as end nodes can make use of a number of tunable policies and parameters.

• **Background Overhead** can be increased to reduce dissemination latency. For example, an unreliable multicast (such as an IP-multicast) could be used to accelerate an insertion request. Gossip would now function just to eliminate gaps in the data replicated at nodes in the system. On the other hand, if a burst of updates occurred, a costly surge in traffic would result.

• **Peer Maintenance** can be done through flexible end-to-end policies, e.g., based on network proximity, preference for peers not connected through a firewall, trusted peers, etc.

• **Multiple tries and Routing of queries** enables a query to reach an appropriate node (i.e., one with a copy of the resource tuple) in the resource's affinity group when the initial single hop lookup fails. TTL (time-to-live) and the number of concurrent retries can be used to trade load and latency against likelihood of success.

For Kelips web caching, the policy choices used are described in Section 3, and Section 4.3 gives experimental results to show the effect of cranking the knobs.

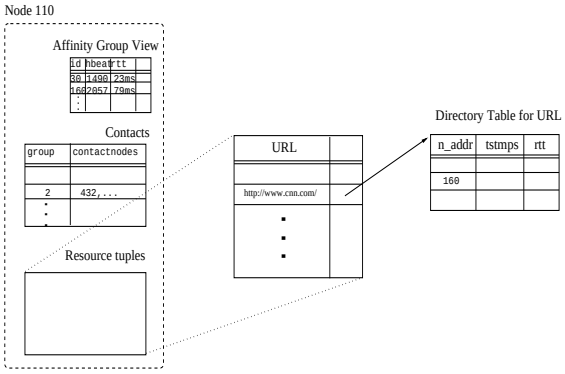

Fig. 2. **Web caching: Modified soft state at a Kelips node.**

3. KACHE: DESIGN OF A P2P WEB CACHING APPLICATION WITH KELIPS

In this section, we discuss the design of Kache, the decentralized web caching application we constructed using Kelips. There are two options to designing an application over a p2p DHT (such as Pastry or Kelips): either (a) layering, through the use of the standard `get(object, ...), put(object, ...)` API exported by the DHT layer (as in [Zhao et al. 2003]), or (b) pushing the application down into the DHT layer. Our work adopts the latter approach (b) since this offers considerable flexibility and performance benefits compared to the approach (a).

The current section describes the required modifications in the Kelips base design - the details of the soft state at each node, the handling of lookups, and finally, where and how the soft state is refreshed. Section 4.3 studies, through cluster-based experiments and trace-based simulations, how well this design supports the initially stated goals for decentralized web caching (viz., tolerance to churn, topologically local access, good hit ratios for low latency and low server bandwidth, and load balancing).

Soft State at A Node. For a web caching application, the system may contain multiple cached copies of a given web object, and hence multiple homenodes for the object. Therefore, Kelips is modified so as to replicate a *directory table* for each object that has cached copies in the system. A directory table is a collection of a small set of addresses of topologically proximate nodes that hold a valid copy of the object. This is depicted in Figure 2.

A directory table has a limited number of entries. Each directory entry contains the following fields: node address *n_addr*; round-trip-time estimate *rtt*; timestamp record *tstmps*. *n_addr* is the address of a node hosting a valid copy of the object; *rtt* is the round-trip-time estimate to this node; *tstmps* is a collection of different timestamps w.r.t. the web object such as time-to-live, time of last modification etc. The *tstmps* fields are used to decide if this copy of the object is fresh at a given point of time.

Web Object Lookup. A request for web object from the browser at a node is handled in the following manner. If a fresh copy of the object exists in the requesting

node's local cache, it is returned to the browser. If a stale copy is found, the node sends a CGET request to one of its contacts for the object's affinity group. If the requesting node has not accessed the object previously, a GET request is sent to one of its contacts for the object's affinity group. The requesting node is itself used as the contact in the case when the requesting node's affinity group is same as that of the object's. At any node n, contacts for a foreign affinity group are maintained using a *peer maintenance* policy that periodically measures the round trip time to contacts, and listens to the membership heartbeat stream seeking to replace the known contact that is farthest from node n with the newly heard-of candidate. Such a peer maintenance policy means that the GET request for an object will be sent to a contact that is topologically nearby to the requesting node.

When the contact receives a CGET request for an object, it first searches for the appropriate directory entry.

If the directory table contains at least one valid entry, the contact forwards the request to the topologically closest node among the entries (using the *rtt* field). This node in turn sends either a *not-modified* message or a copy of the object back to the requesting node. The topological proximity of the contact to both this node and the requesting node ensures access to a nearby cache of the requested object. If the triangle inequality for network distances is satisfied, the distance to the cached copy is at most the sum of the requester-contact and contact-cache distances.

If the directory table contains no valid entries, the contact has two choices - either to return a failure to the requesting node, or to forward the request for object to a peer in its own affinity group. For the former option, the requesting node subsequently contacts the web server directly with a GET/CGET. The latter request forwarding scheme can be generalized to a *query routing* scheme that uses multiple hops for routing a query try and multiple tries per query. The comparative performance of this multi-hop, multi-try (MM) scheme and the basic single-hop (SH) scheme is evaluated experimentally in Section 4.4.

Where Soft State is Maintained and How it is Updated. A given object may have multiple cached copies. Information about a particular cached copy of the object is replicated only *partially* within the object's affinity group, and potentially to nodes that are "nearby" to the homenode of the cached copy. However, this is achieved in a completely decentralized fashion, as described below.

When a node n successfully fetches a copy of an object o not accessed previously by it, the node creates a directory entry $< o, n >$ and communicates it to the contacts for o's affinity group. The contact first searches for object o's directory table, creating one if necessary. If there is an expired duplicate entry for $< o, n >$ already in the table, this is replaced with a fresh entry. Otherwise, if the table is not yet full, a new entry is created for $< o, n >$. Otherwise, the directory table is full since the limit on number of entries is exhausted. In this case, the contact measures the round trip time to node n - if this is less than the highest *rtt* field among directory table entries, the latter entry is replaced by the new $< o, n >$ entry.

Similar to the unmodified Kelips protocol, all object tuples are subject to selection for inclusion in a gossip message, in order to disseminate the new tuple $< o, n >$ within o's affinity group. However, recall from Section 2 that gossip targets are

chosen through a topologically aware distribution (spatial distribution based on round trip times). Thus, gossip messages tend to flow between nodes that are topologically close.

Now, when only a few nodes in the entire system have accessed the given object o, one would ideally want all the nodes in o's affinity group to point to these nodes. However, when the number of cached copies of o rises, and as directory tables begin to fill up, a new tuple $< o, n >$ not previously inserted would replace entries in directory tables of nodes close to node n only. Thus, spreading tuple $< o, n >$ through gossip to nodes that are topologically far from node n will have low utility. This is achieved by associating a hops-to-live *htl* field with the disseminated tuple being disseminated through gossip.

The first contact spreading $< o, n >$ initializes the *htl* field to a small number `HTLMAX` (set to 3 in our experiments). *htl* is decremented at a node if $< o, n >$ is not inserted into the directory table for o. A tuple $< o, n >$ received with $htl = 0$ is not gossiped further.

When there are a large number of clients caching a valid copy of a given object, the effect of the combination of the above scheme and spatial gossiping is twofold. Firstly, the directory entries maintained by a contact are topologically nearby to the contact. This ensures that a requesting node communicating with this contact is potentially also be topologically nearby to the homenode of the cached object. Secondly, as the number of cache copies of a given object rises, the background bandwidth used to propagate information about a new node hosting a copy of the object decreases.

4. EXPERIMENTAL RESULTS

In this section we present the experimental evaluation of Kelips and Kache. We present the results in three parts: We first present the results from the simulation study of Kelips. We then present the microbenchmarks of the core Kelips component of the web caching application running within a commodity PC cluster. Finally, we present the performance evaluation results from a simulation study of Kache.

4.1 Kelips: Simulation Results

Our evaluation was based on a prototype implementation of Kelips, coded in C using the WinAPI. Multiple nodes were run on a single host (1 GHz CPU, 1GB RAM, Win2K) with an emulated network topology layer. The available resources were adequate to let us simulate system sizes of several thousand nodes.

Background overhead in the current configuration consists of one gossip message from each node every 2 (normalized) seconds. Rations limit gossip message size to 272 B. 6 gossip targets are chosen, 3 of them among contacts.

Load Balancing: Resources are inserted into a stable Kelips system. The resource name distribution used is a set of anonymized web URLs obtained from the Berkeley Home IP traces at [InternetTrafficArchive]. The load balancing characteristics are better than exponential (Figure 3). Resource(file) and resource tuple distribution as files are inserted (2 insertions per normalized second of time) is shown in Figure 4; the plot shows that resource tuple distribution has small deviation around the mean.

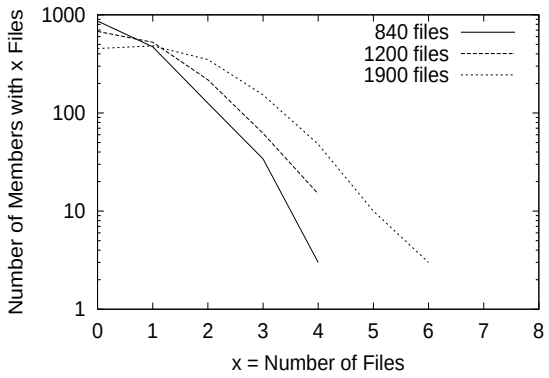

Fig. 3. **Load Balancing I:** *Number of nodes (y-axis) storing given number of files (x-axis), in a Kelips system with 1500 nodes (38 affinity groups).*

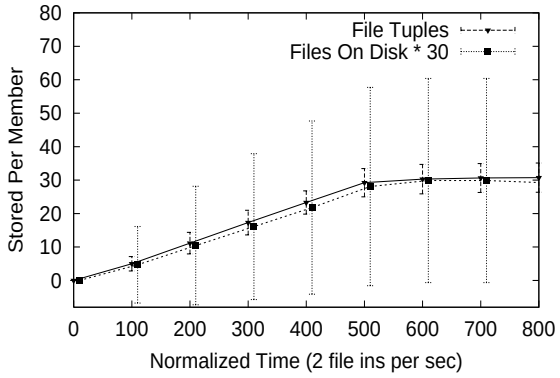

Fig. 4. **Load Balancing II:** *Files are inserted into a 1000 node system (30 affinity groups), 2 insertions per sec between t=0 and t=500. Plot shows variation, over time, of number of files and filetuples at a node (average and one standard deviation).*

Resource Insertion: This occurs through a multi-try (4 tries) and multi-hop scheme (TTL set to $3*logN$ virtual hops). Figure 5 shows the turnaround times for insertion of 1000 different resources. 66.2% complete in 1 try, 33% take 2 tries, and 8% take 3 tries. None fail or require more than 3 tries. Views were found to be fully replicated in this instance. In a different experiment with 1500 nodes and views only 55.8% of the maximum size, 47.2% inserts required 1 try, 47.04% required 2 tries, 3.76% required 3 tries, 0.96% needed 4 tries, and 1.04% failed. Multi-hop routing thus provides fault-tolerance to incompleteness replication of soft state.

Fault-tolerance: Figures 6 and 7 show the fault-tolerance achieved through the use of background overhead (gossip stream). Lookups were initiated at a constant rate and were found to fail only if the homenode had also failed (Figure 6). In other words, multi-hop rerouting and redundant membership information ensures successful lookups despite failures. Responsiveness to failures is good, and mem-

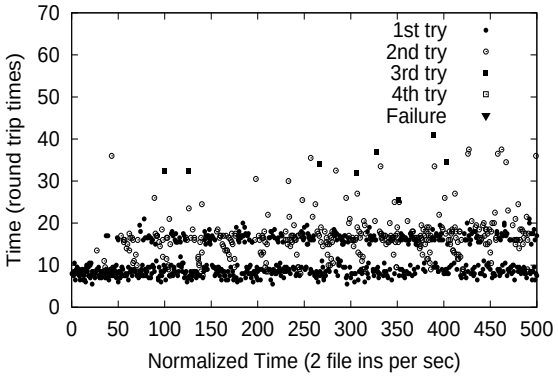

Fig. 5. **File Insertion:** *Turnaround times (in round-trip time units) for file insertion in a 1000-node Kelips system (30 affinity groups).*

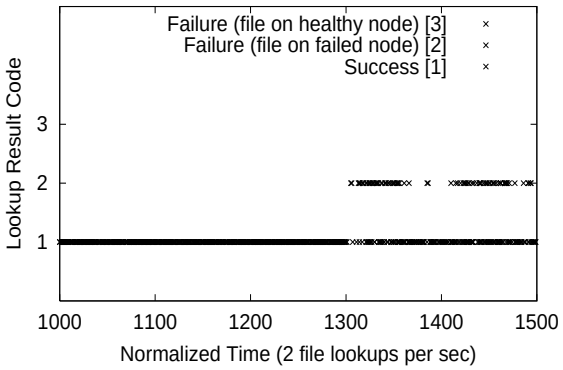

Fig. 6. **Fault Tolerance of Lookups I:** *In a 1000 node (30 affinity groups) system, lookups are generated 2 per sec. At time* $t = 1300$*, 500 nodes are selected at random and caused to fail. This plot shows for each lookup if it was successful* $[y - axis = 1]$*, or if it failed because the homenode failed* [2]*, or if it failed in spite of the homenode being alive* [3].

bership and resource tuple entry information stabilize quickly after a membership change (Figure 7).

4.2 Kelips: Small PC Cluster Results

This section presents microbenchmarks of the core Kelips component of the web caching application running within a commodity PC cluster. The cluster consists of commodity PCs, each with a single CPU (PII or PIII, clock speed ranging from 450 MHz to 1 GHz), RAM size ranging from 256 MB to 1 GB RAM, and running Win2KPro over a shared 100 Mbps ethernet. A single node called the "introducer" is set aside to assist new nodes to join by initializing their membership lists.

We investigate actual memory utilization of the Kelips application and the consistency of membership soft state for a small cluster.

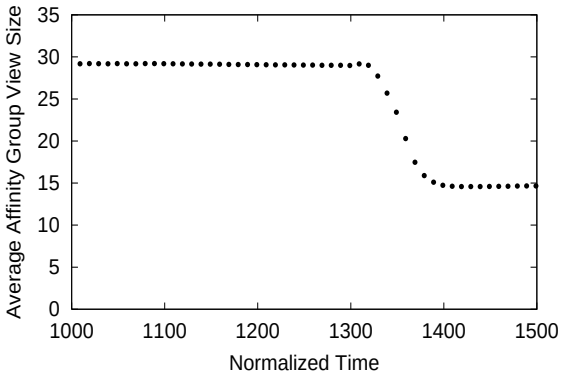

Fig. 7. **Fault Tolerance of Lookups II:** *At time t=1300, 500 out of 1000 nodes in a 30 affinity group system fail. This plot shows that failure detection and view (and hence resource tuple) stabilization occurs by time t=1380.*

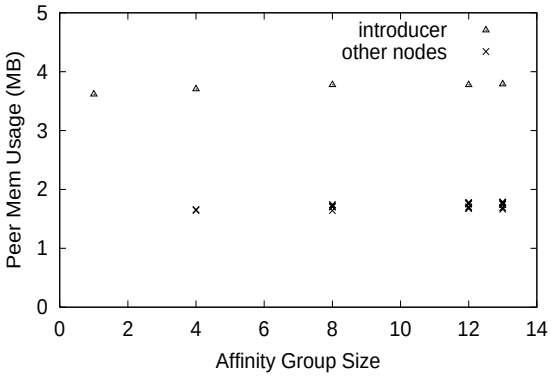

Fig. 8. **Cluster Microbenchmark: Memory Usage of the Kelips Application in a Cluster:** Memory usage in Win2KPro-based hosts, at the introducer node and other nodes.

Memory Utilization. Figure 8 shows the memory utilization at the introducer (triangles) and other nodes (x's) for different group sizes. The base memory utilization is low: less than 4 MB for the introducer at a group size of 1, and less than 2 MB for other nodes at a group size of 4. The rise in memory usage due to an increase in group size is imperceptible for all nodes. We conclude that memory usage in Kelips is modest.

Soft State Consistency. In the experiment of Figure 9, 17 nodes join a one-affinity group system. The background gossiping bandwidth is configured so that at each node, 2 heartbeat entries (each 10 B long) is sent to 5 gossip targets chosen uniformly at random every 2 s. The heartbeat time-out is set to be 25 s. The solid line shows the view size measured at one particular node in the system. The crosses depict the distribution of heartbeat ages received at this node from the gossip stream. The numbers are clustered around less than 10 s for group sizes of

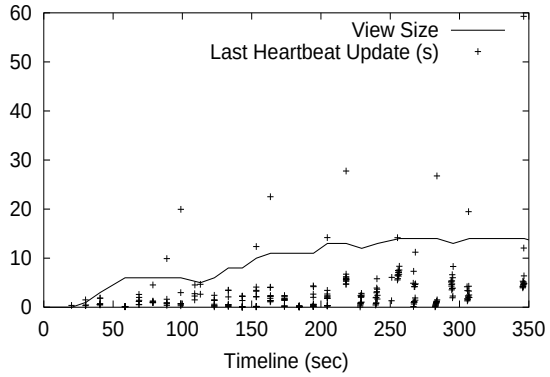

Fig. 9. **Cluster Microbenchmark: Distribution of heartbeat ages and view size at a particular node.**

up to 14. However, there are a few outliers - the ones beyond 25 s lie at times t=220 s, t=290 s, and t=345 s. On closer observation of the solid line, each of these leads to one node being deleted from the view. This explains why there are 14 =(17-3) nodes in the affinity group at time t=350 s.

4.3 Kache: Experimental Evaluation

We evaluate the performance of a C prototype implementation of the Kelips-based web caching system. The evaluation consists of trace-drive experiments to study the system on a larger scale. This study is based on a combination of three traces/maps - client access web traces obtained from the Berkeley Home IP network [Davison], transit-stub network topology maps obtained through the Georgia-Tech generator [GTech], and churn traces from the Overnet deployment (obtained from the authors of [Bhagwan et al. 2003]).

4.4 Trace-Based Experiments

We study the performance of Kelips web caching through trace-based simulations. Multiple client nodes were run on a single host (1 GHz CPU, 1 GB RAM, Win2K) with an emulated network topology layer [5]. The experiments in this section combine three traces - network topologies, web access logs and p2p host availability traces. We enumerate on the first two, and defer a description of the p2p host availability trace until later in the section.

The underlying network topology is generated using the well-known GT-ITM transit stub network model [GTech]. The default topology consists of 3 transit domains, with an average of 8 stub domains each, and an average of 25 routers per stub domain. Each Kelips node is associated with one host, and this host is connected to a router that is selected uniformly at random from among the 600 in the topology. Stubs are connected to each other with probability 0.5, and routers are connected to each other with probability 0.5. Network links are associated with

[5]Limitations on resources and memory requirements restricted current simulation sizes to a few thousand nodes.

Workload Traits	
Number of Clients	916
Total reqs	82142
Total cacheable reqs	75363
Total reqs size	558.9 MB
Total cacheable reqs size	523.3 MB
Total objs	47585
Total cacheable objs	43041
Trace duration	12200 s
Mean req rate	6.73 reqs/s
Perf. of Central Cache	
Total external bandwidth	393.3 MB
Avg. Ext. b/w per req.	4.78 KB
Hit ratio	0.331

Table I. **Workload Characteristics and Centralized Cache Performance on the Berkeley HomeIP web access traces used.**

routing delays, but congestion is not modeled.

The Berkeley HomeIP web access traces [Davison] are used to model object access workloads at Kelips nodes. Each web trace client is mapped to one Kelips node. The characteristics of the traces used are presented in Table I. The last two rows in this table contain numbers corresponding to a single centralized proxy cache with infinite storage. These two numbers are the optimum achievable for this particular trace, with any caching scheme.

Finally, the Kelips group is configured as follows. The default number of participants (nodes) is 1000, and the default number of affinity groups is 31. The single-hop (SH) query routing scheme is the default. Background gossip communication was calculated to consume a maximum of 3 KBps per node. The number of directory entries per web page is limited to 4. We do not limit the cache size at each node, but we study the variation of maximum cache size with time and show that the maximum cache size stays low for the access trace considered.

External Bandwidth. Figure 10 shows, over 500 s intervals, the aggregate bandwidth sent out to web servers due to misses within the p2p web cache. The external bandwidth due to Kelips web caching (dashed line) is comparable to that obtained through a central cache (dotted line).

Hit Ratio. Hit ratio is the fraction of requests served successfully by the p2p cache. Define "oaf" as the number of times an object is accessed throughout the entire trace. As expected, the hit ratio rises with oaf (Figure 11). The plot appears to level out beyond a value of oaf=20.

Single Hop (SH) versus Multihop (MH). In the multi-hop scheme, a request is retried at most 4 times. Out of the four retries at most 2 retries are sent out directly to a contact. The rest are first forwarded to a node in its own affinity group in search of other potential contacts. Each request is routed for at most 3 hops in the requesting nodes affinity group and for at most 3 hops in the target affinity group.

Table II compares the hit ratio and average external bandwidth per request for the single hop (SH) and multi-hop (MH) query routing schemes. A comparison

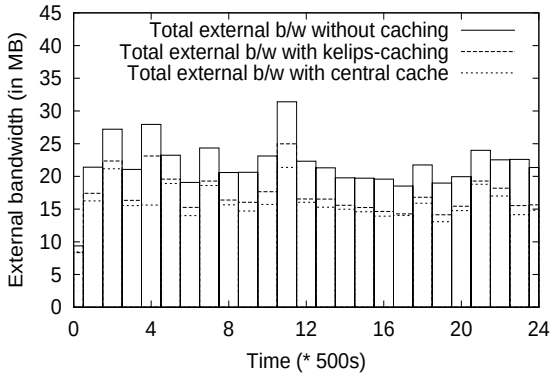

Fig. 10. **External bandwidth vs Time**: Kelips web caching is comparable to that obtained through a central cache.

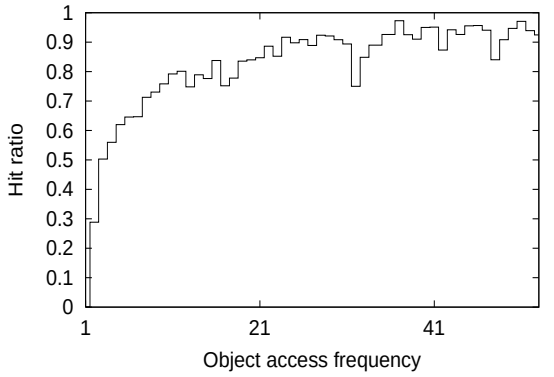

Fig. 11. **Hit ratio vs Object access frequency:** Objects accessed more frequently have higher hit ratios, saturating out at beyond oaf=20.

with Table I shows that the performance of both SH and MM Kelips web caching schemes are only slightly worse than that of the centralized cache scheme. The single hop query routing suffices to achieve as good hit rate as multi-hop, multi-try query routing. The only condition under which MM would be advantageous over SH is if either (a) an insertion of a web object tuple are followed so closely by queries for it (from other nodes) that the resource tuples might not be fully replicated, or (b) high churn rates cause staleness of membership tuples so that the single contact tried by the SH scheme is down. It is evident from Table II that (a) is not true for the web trace workload under study. The reasons why churn rates considered do not affect the hit ratio is explained later in Section 4.4.1.

Access Latency. We measure two types of latency: (a) (*Time to find a target node address*) the time taken to resolve a request and return the address of a cache or report a cache miss to the requesting node, and (b) (*Time to reach a target node*) in the case of an external cache hit, the total time for the request to reach a node

Scheme	Ext. b/w	Hit Ratio
SH	5.63 KB	0.317
MM	5.5 KB	0.323
SH + churn	5.65 KB	0.313
MM + churn	5.46 KB	0.323

Table II. **Average external bandwidth per request and hit ratio or single hop (SH) and multi-hop multi-try (MH) query routing schemes.**

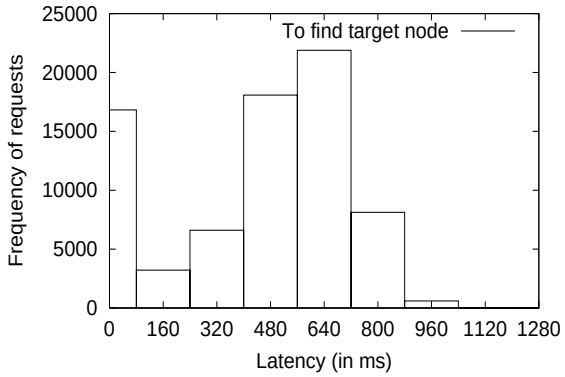

Fig. 12. **Request frequency vs. Latency to search for a cache copy - SH. Plotted for all requests to the external cache.**

with a valid copy of the object (from the time when the request has been first issued at the requesting node). We do not measure the total time to fetch the object as this is a function of the object size. Figure 12 and 13 show these two numbers for the SH query routing scheme. The plots have a bimodal distribution, with a lower peak at a zero latency (local cache hit). Most requests are resolved within 1000 ms, and the total time taken to reach the target cache is within 1200ms for most requests. These plots demonstrate that access latencies are low and confirm the locality awareness of Kelips-caching.

Load Balancing. We investigate the load balancing of requests for web objects in Figure 14. We consider one popular cacheable object. The requests received for this object that are served successfully by the p2p cache system are assigned a global sequence number and plotted on the x-axis. The accessing nodes are ordered globally by their time of access on the y-axis. Each data point (x, y) shows that request number x was served at the node with global sequence number y. If points on this plot were clustered along horizontal lines, it would mean that a few nodes were taking most of the hits. An examination of Figure 14 shows that this is indeed not the case. Kelips web caching thus achieves good load balancing w.r.t. object requests.

Cache Size. Figure 15 shows the variation of cache size with time during the simulation, and validates our infinite cache size assumption since the maximum cache size measured was smaller than 10 MB over the trace of duration 12,200 s.

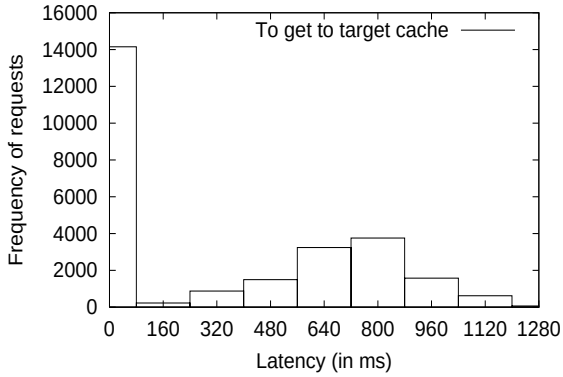

Fig. 13. **Request frequency vs Latency to access cached copy - SH. Plotted for requests that result in external cache hits.**

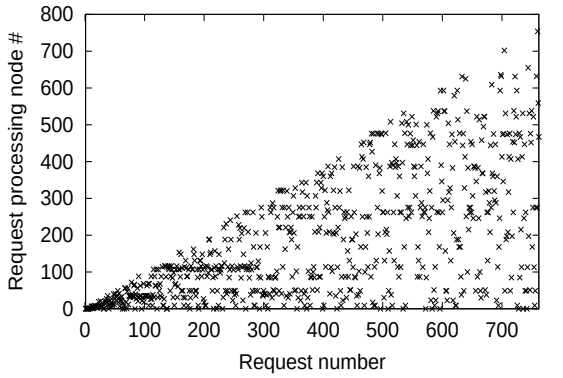

Fig. 14. **Req processing node# vs Req num (see text in "Load Balancing" for explanation)**

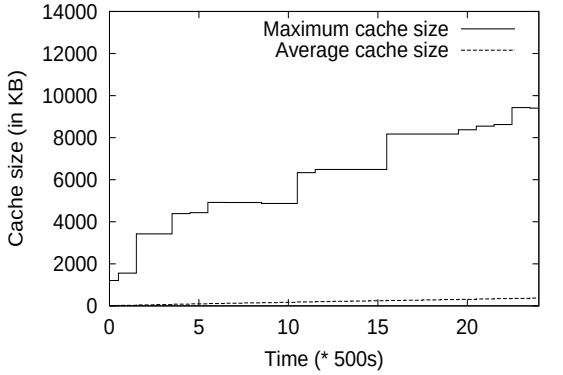

Fig. 15. **Cache size vs Time:** Average and Maximum cache sizes are smaller than 10 MB throughout the trace of duration 12,200 s.

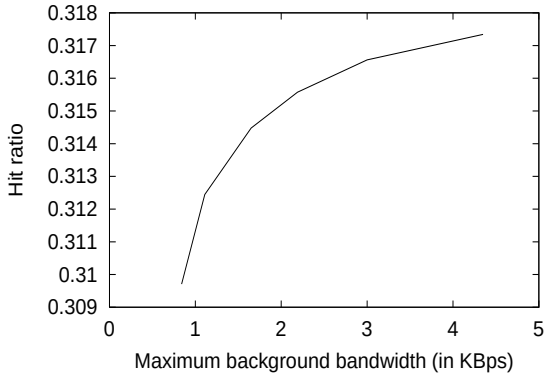

Fig. 16. **Hit ratio vs Max background bandwidth:** Increased background gossip communication cost affects the hit ratio by increasing the number of fresh web object tuples.

Background Bandwidth. We investigated the effect of varying the background gossip bandwidth (i.e, bandwidth used at end nodes) on the performance of the web caching scheme. We observe from Figure 16 that hit ratio decreases with decreasing background bandwidth since web object tuples are replicated less widely, and thus fewer queries hit a node with fresh tuples. Yet, the decrease is not substantial - from 4.35 KBps to 0.84 KBps, the hit ratio decreases by 0.005.

4.4.1 *Effect of Churn: Constant Node Arrival and Departure Rates.* The experiments presented above in section studied the effect of multiple node failures, measured the time for membership convergence, and showed that Kelips continues to ensure that lookups succeed efficiently under such stresses. In this section, we study the effects of a more general class of stresses arising from "churn" in the system - rapid arrival and failure (or departure) of nodes - on our implementation of web caching.

Our study uses client availability traces from the Overnet p2p system, obtained through the authors of reference [Bhagwan et al. 2003]. These traces specify at hourly intervals which clients (from a population of 990) are logged into the system. Typically, about 20% of the 990 clients are up at the start of each hour, and the hourly turnover rate varies between 10% - 25% of the total number of clients that are up.

Effect on Membership. Each Kelips node in a 990-node system (with 31 affinity groups) is mapped to a node in this trace. Hourly availability traces are then injected into the system periodically at the start of *epochs* (rather than continuously) - given the hourly availability traces, this injection models the worst case behavior of Kelips from the churn.

Figure 17 shows the average affinity group view size when a new churn trace is injected every 200 s (in other words, 1 hour in the availability traces is mapped to 200 s). This epoch is more than the average stabilization period of the current Kelips configuration. As a result, one sees that soon after the trace injection at the beginning of an epoch, there is first a surge in membership size as information about returning nodes is spread through the system. This is followed by an expiry

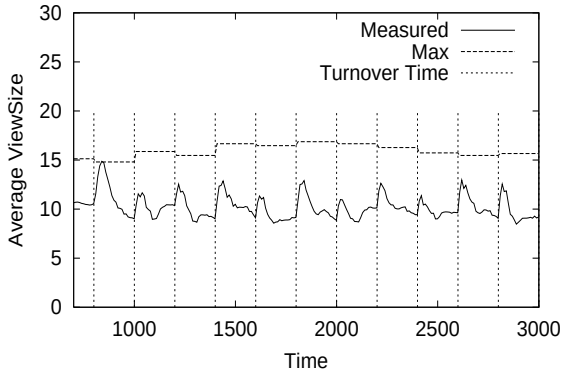

Fig. 17. **Effect of churn on Affinity Group View Size at a node:** Hourly availability traces from the Overnet system are periodically injected into the system (at the times shown by the vertical bars). Churn trace injection epoch for this plot is 200 s.

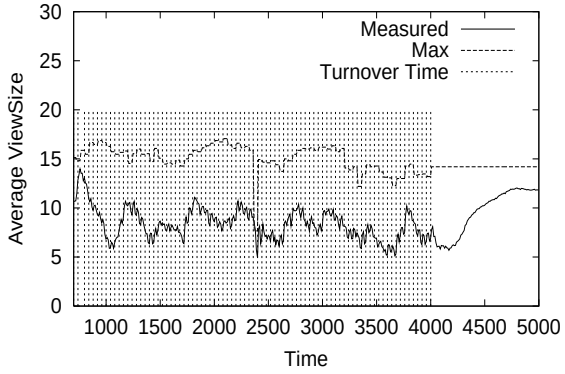

Fig. 18. **Effect of churn on Affinity Group View Size at a node:** Hourly availability traces from the Overnet system are periodically injected into the system (at the times shown by the vertical bars). Churn trace injection epoch for this plot is 40 s.

of nodes that have become unavailable due to the trace injection. In most epochs, the membership stabilizes a little before the end of the start of the next epoch [6].

Figure 18 shows the same experiment with churn traces injected every 40 s. The effect of such a low injection epoch is dramatic – the system suffers considerable pressure and is unable to cope with rapid membership changes. Before the membership changes from the last trace injection can be spread or detected by the system, a new trace is injected. As a result, the size of the membership lists thrash. Even after churn traces have been stopped being injected at time t=4000 s, the system takes considerable time to recover.

[6]The epoch starting at 1800 s is an exception. In this case 200 s was not quite enough time for the system to stabilize.

Effect on Hit Ratio, Access Latency. Deployments of peer-to-peer applications tend to invite both nodes that are long lived and thus available most of the time, as well as nodes that exhibit churn behavior [Saroiu et al. 2002]. For this experiment, we choose an operation point where 50% of the nodes in the Kelips system are available, and the remaining 50% are churned. More specifically, in a system of 1000 nodes, 500 nodes were churned by mapping to the first 500 entries in the Overnet availability traces[7]. The default churn trace injection epoch was set to 200 simulation time units. The other 500 nodes were kept alive throughout the trace, and requests were issued to the trace through these.

Figures 19 and 20 show the request latency distributions under the effect of churn. A comparison with Figures 12 and 13 respectively, and a glance at Table II, show that churn has an *a negligible effect on the hit ratio and access latency distributions.* This happens in spite of membership tuples varying as shown in Figure 17, and the use of only single hop (and not multi-hop multi-try) query routing.

This churn-resistant behavior arises from the proactive contact maintenance policies used in Kelips. Recollect that when a Kelips node hears about another node in a foreign affinity group, it uses this node to replace the farthest known contact for the foreign affinity group. In addition, recollect that when a contact entry expires (as might happen when the contact node is being churned), the expired entry is retained for a time duration to prevent stale copies for that node from being reinserted into the contact list within the specified timeout. Since the retention timeout is set to an excess of 200 time units in this experiment, the above two algorithms result in each Kelips node settling on a set of contacts that are nearest to it, as also highly available (not churned). Queries thus get routed mostly among the nodes that are stable, thus succeeding as often as in the simulation runs without churned nodes.

Figure 21 shows that hit ratio decreases by an insignificant amount (0.006, 2% decrease) as the churn trace injection epoch is decreased from 240 s to 20 s. Even when affinity group membership entries are thrashing at a churn trace injection epoch of 40 s (as shown in Figure 18), the hit rate is 30.9%, only 0.4% below the hit rate with a churn trace injection epoch of 200 time units. The reasoning behind this plot follows along the same lines as in the previous paragraph.

Note from Figure 13 that the number of requests which are local hits is about 18.8% of all the cacheable requests. Although a large portion of the cache hits from Figure 13 (54.4%) are local, we focus on the stability of the non-local p2p cache hits (the "remaining 45.6%"). From Figure 21, we see that a large fraction of these hits are retained even when there is excessive churn in the system.

This study thus substantiates our claim that Kelips web caching survives high rates of churn attack on the system.

[7]This is justified by the results of [Bhagwan et al. 2003] showing that availability characteristics tend to be uncorrelated across clients.

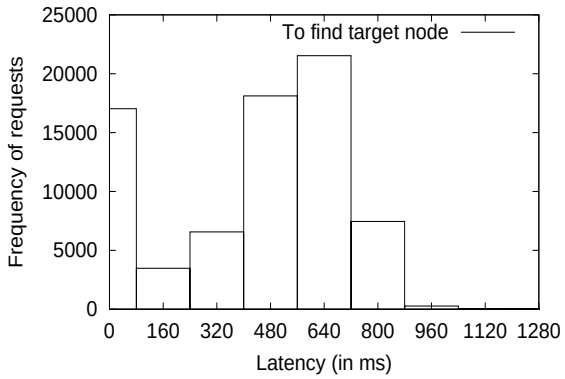

Fig. 19. **Effect of Churn: Request frequency vs. Latency to search for a cache copy - SH. Plotted for all requests to the external cache. Churn trace injection epoch is 200 s.**

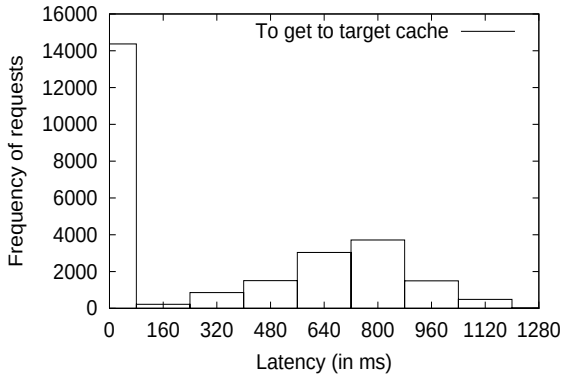

Fig. 20. **Effect of Churn: Request frequency vs Latency to access cached copy - SH. Plotted for requests that result in external cache hits. Churn trace injection epoch is 200 s.**

5. RELATED WORK

5.1 Web Caching Schemes

Normal HTTP Request Processing. A client's request for a web object is first serviced from the local cache on the client's machine. This might fail because either the object is uncacheable, or not present in the cache, or the local copy is stale [8]. In the first case, the object's web server is contacted by issuing an HTTP GET application level request. In the second case (client cache miss), an HTTP GET is issued to the external web cache. For the third case (client cache copy stale), an HTTP conditional CGET is issued to the external web cache. If the external

[8]Freshness is determined through the use of an expiration policy in the web cache. The expiration time is either specified by the origin server or is computed by the web cache based on the last modification time.

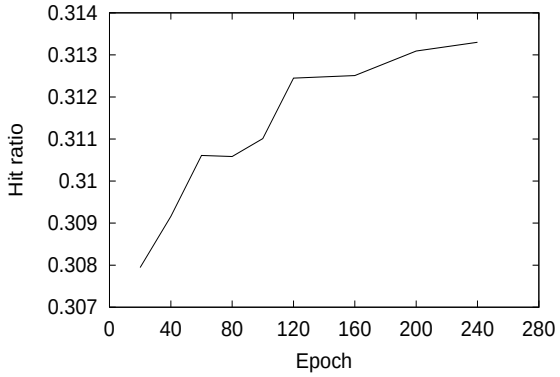

Fig. 21. **Effect of Churn: Hit rate vs Churn trace injection epoch.**

web cache is unable to service the GET (CGET), it could either fall-through to the web server or inform the client to contact the server directly. The reply to an external web cache request is the object or, in case of a CGET, a *not-modified* reply indicating that the stale copy is indeed the latest version of the object.

The design of an external web cache falls into one of the following three categories : (1) a hierarchy of proxies, (2) distributed proxies, or (3) peer-to-peer caches. We present a truncated survey below - the interested reader is referred to [Wang 1999] for a comprehensive study.

1. Hierarchical Schemes: Harvest [Chankhunthod et al. 1996] and Squid [Wessel] connect multiple web proxy servers at the institutional, wide area network and root levels in a virtual hierarchy. Servers store caches of objects, and an external web cache request is serviced through these multiple levels by traversing the parent, child and sibling pointers. Chankhunthod et al [Chankhunthod et al. 1996] found that up to three levels of proxy servers could be maintained without a latency loss compared to that of direct web server access. Wang [Wang 1999] outlines some of the drawbacks of the hierarchy - proxy placement, redundant cache copies, and load on servers close to the root, etc.

2. Distributed Caching: Provey and Harrison [Provey and Harrison 1997] store only cache hints (not objects) at proxy servers. Cachemesh [Wang and Crowcroft 1997] partitions out the URL space among cache servers using hashing. A cache routing table among the servers is then used to route requests for objects.

3. Peer-to-peer Caching: The above schemes still require a proxy infrastructure. The elimination of proxy servers completely implies that the meta-information that would normally be stored inside the hierarchy must instead be stored at the individual clients or the server.

Padmanabhan et al [Padmanabhan and Sripanidkulchai 2002] examine a server redirection scheme that uses IP prefixes, network bandwidth estimates, and landmarks to redirect a client request at the web server to a nearby client. Peer-to-peer web caching schemes such as COOPnet, BuddyWeb, Backslash and Squirrel organize network clients in an overlay within which object requests are routed. Stading et al [D. Liben-Nowell 2002] propose institutional level special DNS and HTTP

servers, called "Backslash" nodes. Backslash nodes are organized within the Content Addressable Network (CAN) overlay, and an external web cache request is routed from a client to the nearest Backslash node, and then into the CAN overlay itself. BuddyWeb [Wang et al. 2002] uses a custom p2p overlay among the clients themselves to route object requests. Squirrel [Iyer et al. 2002] builds a cooperative web cache on top of the Pastry p2p routing substrate.

Padmanabhan et al contended in [Padmanabhan and Sripanidkulchai 2002] that the use of peer-to-peer routing substrates for web caching may be too "heavy-weight because individual clients may not participate in the peer to peer network for very long, necessitating constant updates of the distributed data structures". Work on the p2p cooperative web cache designs described above has not addressed this criticism. Although the above p2p overlays are self-reorganizing, we believe our paper is the first systematic study of cooperative caching under the form of "churn attack" discussed earlier.

There is a preliminary theoretical study by the authors of article [Liben-Nowell et al. 2002] on how the Chord peer-to-peer system uses a periodic stabilization protocol to combat the effect of concurrent node arrival and failure. Their theoretical analysis however revealed that such a protocol would be infeasible to run - either the time to stabilization or the bandwidth consumed grow super-linearly with the number of nodes. The Kelips web caching solution does not require supplementary stabilization protocols; constant-cost and low-bandwidth background communication suffices to combat significant rates of churn while ensuring favorable and robust performance numbers. Our study in the current paper is also the first to demonstrate a practicable and efficient solution to the problem of churn and experimentally study its working under realistic conditions.

5.2 Peer-to-Peer systems

Napster [OpenNap] is the one that started it all. Napster is the first p2p file sharing system. Lookups in Napster were resolved at a central node and hence Napster was not fully decentralized. Gnutella [Gnutella] is one of the few that followed. Gnutella falls under the category of *unstructured* P2P systems and relies on flooding. Every node/peer exports the data it wants to share with other peers in the system. Each request is associated with a TTL and the request is forwarded to all neighbors of the initiator node. Every node receiving a request decrements the TTL and forwards the request to its neighbors. Forwarding of the request continues as long as TTL is positive. Lookup time here is reasonable (logarithmic) but the bandwidth requirements are tremendous. FreeNet [Wiley et al. 2000] and Routing Indices [Crespo and Garcia-Molina 2002] are other examples of *unstructured* indices.

DHTs are examples of *structured* indices. Chord [Dabek et al. 2001], Pastry [Rowstron and Druschel 2001], Tapestry [Zhao et al. 2001], CAN [Ratnasamy et al. 2001], Viceroy [Malkhi et al. 2002], Kademlia [Kaashoek and Karger 2003a] and Koorde [Kaashoek and Karger 2003b] are some of the examples of other DHTs proposed in the literature. Chord is a simple index structure that ensures that worst case search cost is logarithmic in the number of peers. Peer addresses and keys are hashed to the same identifier space(say, 0 to $2^m - 1$). A key is stored with the first peer having a peer id equal to or greater than the key id. Each peer has pointers to its successor and its predecessor and also to peers which are 2^i ($i = 1$

to $m-1$)hops away on the identifier space. Chord and other similar DHTs as we already pointed out do not handle churn well. Kelips on the other hand is a one-hop DHT that has the "flexibility" to allow for robustness to churn.

6. CONCLUSION

We have shown how to design a churn-survivable and locality-adaptive peer-to-peer application. Our study has focused on the caching of web objects, and our solution has relied on the use of probabilistic techniques in the framework of the Kelips peer-to-peer overlay (DHT). Evaluation through microbenchmarking on commodity clusters, as well as experiments done through a combination of web access logs, transit-stub topologies, and p2p host availability traces, reveal significant advantages of locality and load balancing over previous designs for p2p web caching. Hit ratios and external bandwidth usage are both comparable to that in centralized web caching, even when the system is subjected to high rates of churn. In a system with a 1000 nodes, background communication costs as low as 3 KBps per peer suffice to ensure favorable and stable hit ratio, latency, external bandwidth use, and load balancing for access of web objects in the presence of system churn that causes 10%-25% of the total number of nodes to turn over within a few tens of seconds.

The investigation in this paper can be extended to studies in several interesting directions - (1) the hit ratio and latency behavior of Kelips web caching at other operation points than the "50% available - 50% churned" above, (2) the effect of churn on caching scenarios other than web page browsing, and (3) the feasibility of the Kelips constant-cost low-bandwidth solution to other applications and other stressful networking environments.

REFERENCES

Bailey, N. 1975. *Epidemic Theory of Infectious Diseases and its Applications*. Hafner Press.

Bhagwan, R., Savage, S., and Voelker, G. 2003. Understanding availability. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*. 135–140.

Birman, K., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. 1999. Bimodal multicast. *ACM Transactions on Computer Systems 17,* 2 (may), 41–88.

Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M. F., and Worrell, K. J. 1996. A hierarchical internet object cache. In *Proc. 1996 Usenix Technical Conference*. San Diego, CA.

Crespo, A. and Garcia-Molina, H. 2002. Freenet: A distributed anonymous information storate and retrieval system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*.

D. Liben-Nowell, H. Balakrishnan, D. K. 2002. Observations on the dynamic evolution of peer-to-peer networks. In *Proc. 1st International Workshop Peer-to-Peer Systems (IPTPS), LNCS 2429*. Springer-Verlag.

Dabek, F., Brunskill, E., Kaashoek, M. F., and Karger, D. 2001. Building peer-to-peer systems with chord, a distributed lookup service. In *Proc. 8th Wshop. Hot Topics in Operating Syst., (HOTOS-VIII)*.

Davison, B. D. Web caching and content delivery resources. `www.web-caching.com`.

Demers, A., Greene, D., Hauser, J., Irish, W., and Larson, J. 1987. Epidemic algorithms for replicated database maintenance. In *Proc. 6th ACM Symp. Principles of Distributed Computing (PODC)*. 1–12.

Gnutella. Web site. `http://www.gnutella.com/`.

GTech. Modeling topology of large internetworks. http://www.cc.gatech.edu/projects/gtitm.

Gupta, I., Birman, K., Linga, P., Demers, A., and van Renesse, R. 2003. Kelips: building an efficient and stable p2p dht through increased memory and background overhead. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*. 81–86.

InternetTrafficArchive. Web site. http://ita.ee.lbl.gov.

Iyer, S., Rowstron, A., and Druschel, P. 2002. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*.

Kaashoek, M. F. and Karger, D. R. 2003a. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'2003)*.

Kaashoek, M. F. and Karger, D. R. 2003b. Koorder: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'2003)*.

Kempe, D., Kleinberg, J., and Demers, A. 2001. Spatial gossip and resource location protocols. In *Proc. 33rd ACM Symp. Theory of Computing (STOC)*. 163–172.

Liben-Nowell, D., Balakrishnan, H., and Karger, D. 2002. Observations on the dynamic evolution of peer-to-peer networks. In *Proc. 1st International Workshop Peer-to-Peer Systems (IPTPS), LNCS 2429*. Springer-Verlag.

Linga, P., Gupta, I., and Birman, K. 2003. A churn-resistant peer-to-peer web caching system. In *Proc. 1st Workshop on Survivable and Self-Regenerative Systems (SSRS)*.

Malkhi, D., Naor, M., and Ratajczak, D. 2002. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of Distributed Computing (PODS'02)*.

Mohan, C. 2002. Caching technologies for web applications. Talk at Cornell University, Ithaca, NY. http://ita.ee.lbl.gov.

OpenNap. Web site. http://opennap.sourceforge.net.

Padmanabhan, V. N. and Sripanidkulchai, K. 2002. The case for cooperative networking. In *Proc. 1st International Workshop Peer-to-Peer Systems (IPTPS), LNCS 2429*. Springer-Verlag.

Provey, D. and Harrison, J. 1997. A distributed internet cache. In *Proc. 20th Australian Computer Science Conference*. Sydney, Australia.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. 2001. A scalable content addressable network. In *Proceedings of the ACM SIGCOMM'01 Conference*. San Diego, California.

Rowstron, A. and Druschel, P. 2001. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*.

Saroiu, S., Gummadi, P., and Gribble, S. 2002. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking (MMCN)*.

van Renesse, R., Minsky, Y., and Hayden, M. 1998. A gossip-style failure detection service. In *Proceedings of IFIP Middleware*.

Wang, J. 1999. A survey of web caching schemes for the internet. *ACM Computer Communication Review 29,* 5 (oct), 36–46.

Wang, X. Y., Ng, W. S., Ooi, B. C., Tan, K. L., and Zhou, A. Y. 2002. Buddyweb: a p2p-based collaborative web caching system. In *Proc. International Workshop on Peer-to-Peer Computing*.

Wang, Z. and Crowcroft, J. 1997. Cachemesh: a distributed cache system for the world wide web. In *Proc. Web Cache Workshop*.

Website. Fireflies of selangor river, malaysia. www.firefly-selangor-msia.com/fabout.htm.

Wessel, D. Squid internet object cache. http://squid.nlanr.net.

Wiley, B., Clarke, I., Sandberg, O., and Hong, T. W. 2000. Freenet: A distributed anonymous information storate and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*.

Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. 2001. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. In *Technical Report UCS/CSD-01-1141, University of California at Berkeley.*

Zhao, F. D. B., Druschel, P., Kubiatowicz, J., and Stoica, I. 2003. Towards a common api for structured peer-to-peer overlays. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS).*

Received June 2004; revised Month Year; accepted Month Year