A Churn-Resistant Peer-to-Peer Web Caching System

Prakash Linga
Cornell University
Ithaca, NY 14853
linga@cs.cornell.edu

Indranil Gupta
Univ. of Illinois,
Urbana-Champaign
Champaign, IL 61801
indy@cs.uiuc.edu

Ken Birman[*]
Cornell University
Ithaca, NY 14853
ken@cs.cornell.edu

ABSTRACT

Denial of service attacks on peer-to-peer (p2p) systems can arise from sources otherwise considered non-malicious. We focus on one such commonly prevalent source, called "churn". Churn arises from continued and rapid arrival and failure (or departure) of a large number of participants in the system, and traces from deployments have shown that it can lead to extremely stressful networking conditions. It has the potential to increase host loads and block a large fraction of normal insert and lookup operations in the peer-to-peer system. This paper studies a cooperative web caching system that is resistant to churn attacks. Based on the Kelips peer-to-peer routing substrate, it imposes a constant load on participants and is able to reorganize itself continuously under churn. Peer pointers are automatically established among more available participants, thus ensuring high cache hit rates even when the system is stressed under churn. In addition, the system improves on the network locality of cache accesses in previous web caching schemes. The paper presents experimental results from a real implementation running over a commodity PC cluster, as well as trace-based simulations that use real host availability traces obtained from another deployed p2p system.

1. INTRODUCTION

Many systems today are built upon the assumption that if they are secure and available, they are robust. However, security violations such as denial of service attacks can be initiated by sources otherwise considered non-malicious. We consider one such source of possible disruptions in the service of a peer-to-peer system - churn and continuously occurring failures of nodes and communication. We investigate this within the context of a peer to peer system for cooperative caching of web services and objects, and show that probabilistic techniques can be used to build a design that heals 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.

An *external caching* scheme stores copies of web objects or meta-data so that clients can avoid requests to a web server. The best known example of an external cache is a web proxy, but with the rapid spread of web technologies, many other caching scenarios can be identified. Caching is central to performance in the web, both because it reduces load on servers, and because cached data is normally cheaper to access from another machine that is closer in the network. Although cooperation between cache managers is not a common feature of existing web architectures, cooperative caching supported by peer-to-peer architectures has great potential [9, 13]. When sets of client systems have similar interests and fast low-latency interconnections, shared caching could bring significant benefits.

The peer-to-peer mechanisms referred to above emerged from interest in file sharing, and they are a technique with applicability to sharing web caches [9, 13, 15]. We focus on cached web pages, primarily because detailed traces are available for this case and because such an analysis permits direct comparison with other systems.

However, cooperative caching is more generally applicable, and it could improve performance for web services and applications. Mohan studies a variety of these scenarios in [12]. For example, dynamic web content serving can be accelerated by caching html fragments, web applications such as 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 [13]. 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 system [6], is unusual in employing probabilistic schemes and a self-regenerating data structure that

[*]The authors were supported in part by DARPA/AFRL-IFGA grant F30602-99-1-0532 and in part by a MURI grant AFOSR F49620-02-1-0233, with additional support from the AFRL-IFGA Information Assurance Institute, from Microsoft Research and from the Intel Corporation.

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 focuses on server-bandwidth, lookup time and access latency both when a cache hit occurs and when a miss is detected, load balancing and robustness to failures and churn. Our work is experimental, and includes (a) a microbenchmark study for small clusters, and (b) a trace-drive simulation study for larger-sized systems. Our evaluation uses web access traces from the Berkeley Home IP network [18], transit-stub network topology maps obtained through the Georgia-Tech generator [19], and churn traces from the Overnet deployment (obtained from the authors of [2]). 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.

2. RELATED WORK

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 [1]. 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 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 [16] for a comprehensive study.

1. Hierarchical Schemes: Harvest [4] and Squid [17] 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 [4] 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 [16] 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 [20] store only cache hints (not objects) at proxy servers. Cachemesh [21] 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 [13] 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 [22] 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 [15] uses a custom p2p overlay among the clients themselves to route object requests. Squirrel [9] builds a cooperative web cache on top of the Pastry p2p routing substrate.

Padmanabhan et al contended in [13] 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 [11] 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.

3. THE KELIPS PEER-TO-PEER OVERLAY

Peer-to-peer overlays for object insertion and retrieval such as Pastry, Tapestry, and Chord define how each participant node chooses peers to which they maintain pointers. In contrast, Kelips uses softer rules to determine sets of possible peer pointers, permitting a node to pick any peer within the set according to end-user considerations and constraints such as topology awareness, trust, security concerns, etc. The choice can vary over time, and this gives Kelips the

[1]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.

"self-regenerating" behavior mentioned earlier.

More precisely, a Kelips system with n nodes consists of $\sqrt{n}$ virtual subgroups called *affinity groups*, numbered 0 through $(\sqrt{n} - 1)$. 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 $[0, \sqrt{n} - 1]$. Using SHA-1 as the hash function, each affinity group size will lie in an interval around $\sqrt{n}$ w.h.p. The value of n should be consistently known at all nodes, but can be an estimate of the actual system size.

At each node, Kelips replicates resource tuples and membership information. Membership information includes (a) the affinity group *view*, a (partial) set of other nodes lying in the same affinity group, and (b) for each foreign affinity group, a small (constant-sized) set of *contact* nodes lying in it. Each membership entry (affinity group view or contact) carries additional fields such as round-trip time estimates and heartbeat counts.

A node inserting a resource tuple {`resource_name, location`} first determines the resource's affinity group by hashing the `resource_name`. The tuple is communicated to a contact in the resource's affinity group, which in turn disseminates it, perhaps partially, within the affinity group. With full tuple replication, a node can access a resource tuple given a resource's name by one RPC to its contact for the resource's affinity group.

The freshness of resource tuples is determined through the use of an integer heartbeat count. The inserting node periodically disseminates an updated heartbeat count to the resource's affinity group. Similarly, each node n_i periodically disseminates a heartbeat count for itself, refreshing membership entries that are maintained for n_i at other nodes in the system. Thus, failure of n_i leads to purging of membership tuples for n_i (and associated resource tuples with `location` $= n_i$) at other nodes.

Membership heartbeat counts need to be disseminated throughout the entire system (since contacts are maintained across affinity groups). So does membership information such as about joining, leaving or failed members.

All dissemination in Kelips occurs through a continuously active, low-cost background communication mechanism based on a peer-to-peer epidemic-style (or *gossip-style*) protocol [7, 8]. A faster scheme might employ IP multicast for this purpose, using gossip only to fill gaps, but exploration of that option is outside the scope of this paper.

Gossip-based communication in Kelips proceeds as follows. A node periodically picks a few peers (from among its list of contacts and its affinity group view) as gossip targets. These peers are picked based on a spatial distribution based on round trip times [10], that as a consequence, prefers gossip targets topologically close to the node. The node then sends these nodes a constant sized gossip message (via UDP) containing membership and resource tuple entries selected uniformly at random from among those it maintains. These gossiped entries also contain the corresponding heartbeat counts. Tuples that are new or were recently deleted are explicitly added on to the gossip message for faster dissemination. Recipients update their soft state based on information obtained from the gossip message. No attempt is made to detect or resend lost messages.

The latency of dissemination within an affinity group depends on the gossip target selection scheme – it varies as $O(log^2(n))$ under the spatial gossiping scheme from [10] and as $O(log(n))$ under uniform target selection. These latencies rise by a factor of $O(log(n))$ for dissemination throughout the system (i.e., across affinity groups) - see [6]. Since gossip message sizes are limited, only a part of the soft state can be sent with each gossip message. This imposes an extra multiplicative factor of $O(\sqrt{n})$ for heartbeat updates. In fact, however, the constants are small for medium sized systems. In a system with a thousand nodes, a background bandwidth utilization of a few KBps per node suffices to have low dissemination latency ranges in tens of seconds [6].

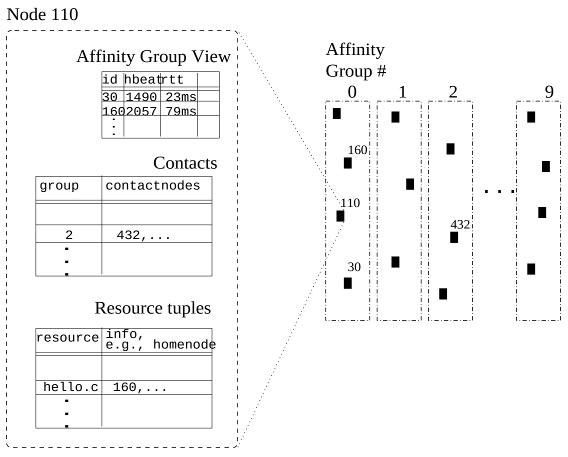

Figure 1: Kelips soft state at a node: A Kelips system with nodes distributed across 10 affinity groups, and soft state at a hypothetical node.

When a membership tuple expires, the deleted entry is retained for a while in order to prevent stale copies for the failed node from being reinserted. This strategy will assume importance later when we discuss the churn-resistance of Kelips web caching.

The above soft state occupies a small memory footprint at each Kelips node. Reference [6] calculates that in a system with 10 million files and a 100,000 nodes, the soft state at each node is 1.93 MB. Section 5.1 of the current paper shows that the implementation occupies a small footprint in a Windows runtime environment.

Figure 1 illustrates an example of a base Kelips system. Membership and resource tuple entries are stored in AVL tree structures to support efficient operations.

Kelips Flexibility: While designing a Kelips-based p2p application (such as web caching), the designer as well as end nodes are equipped with a flexible choice of policies and tuning knobs.

- **Background Overhead** can be increased to lower dissemination latency.
- **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 it to reach an appropriate node (i.e., one with a copy of the resource tuple) in the resource's affinity group when the 1 RPC lookup fails. TTL (time-to-live) and the number of retries can be used to trade latency with likelihood of success.
- **Replication Policies** for the resource (not the resource tuple) can be chosen orthogonal to the base operation of

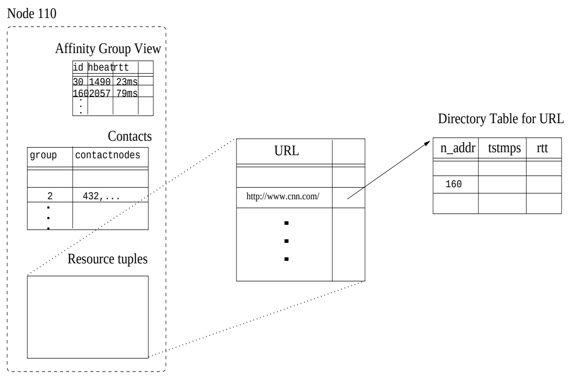

Figure 2: Web caching: Modified soft state at a Kelips node.

Kelips.

For Kelips web caching, the policy choices used are described in Section 4, and Section 5 gives experimental results to show the effect of cranking the knobs.

4. DESIGN OF A P2P WEB CACHING APPLICATION WITH KELIPS

We study the design of a decentralized web caching application with Kelips. There are two options to designing an application over a p2p DHT (such as Pastry or Kelips) : (a) layering, through the use of the standard `get(object, ...)`, `put(object, ...)` API exported by the DHT layer (as in [5]), and (b) pushing the application down into the DHT layer. Our work adopts the latter. 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 5 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 our web caching application, Kelips is used to replicate a *directory table* for each cached object. 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.

Concretely, a directory entry contains the following fields: node address *n_addr*; round-trip-time estimate *rtt*; timestamp record *tstmps*. *n_addr* is the address of the 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 constantly 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 5.2.

Where Soft State is Maintained and How it is Updated. 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 the table is empty, a new entry is created for $< o, n >$. An expired duplicate entry for $< o, n >$ is replaced by a new one. If the table is full, the contact measures the round trip time to node n - if this exceeds the highest *rtt* field among directory table entries, the latter entry is replaced by a 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 3 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. 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.

5. EXPERIMENTAL EVALUATION

We evaluate the performance of a C prototype implementation of Kelips web caching. The evaluation consists of (a) cluster-based microbenchmarks to examine the memory usage of the application and the soft state consistency, and (b) trace-drive experiments to study the system on a larger scale. The latter study is based on a combination of three traces/maps - client access web traces obtained from the Berkeley Home IP network [18], transit-stub network topology maps obtained through the Georgia-Tech generator [19], and churn traces from the Overnet deployment (obtained from the authors of [2]).

5.1 Microbenchmarks: Small PC Cluster

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 450 MHz - 1 GHz CPUs (PII or PIII), 256 MB - 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.

Memory Utilization. Figure 3 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 4, 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 timeout 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

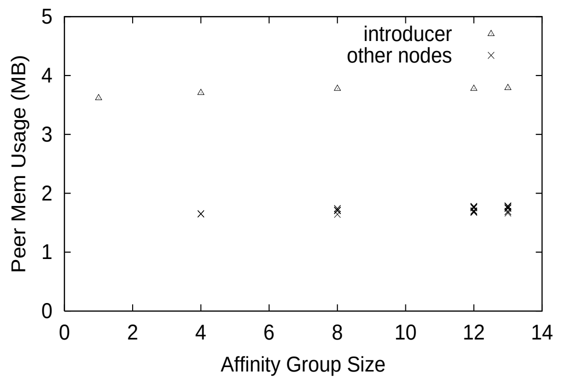

Figure 3: Cluster Microbenchmark: Memory Usage of the Kelips Application in a Cluster: Memory usage in Win2KPro-based hosts, at the introducer node and other nodes.

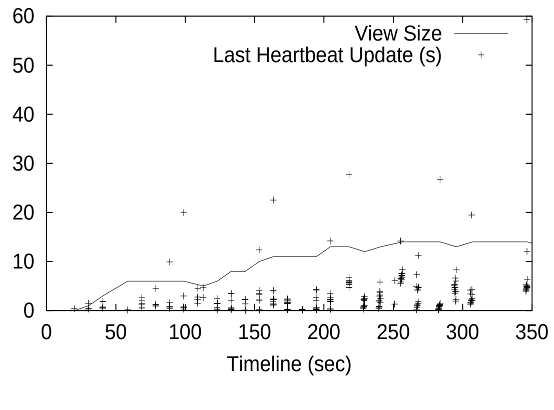

Figure 4: Cluster Microbenchmark: Distribution of heartbeat ages and view size at a particular node.

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 1: Workload Characteristics and Centralized Cache Performance on the Berkeley HomeIP web access traces used.

around less than 10 s for group sizes of 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.

5.2 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 [2]. 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 [19]. 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 routing delays, but congestion is not modeled.

The Berkeley HomeIP web access traces [18] 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 1. 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

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

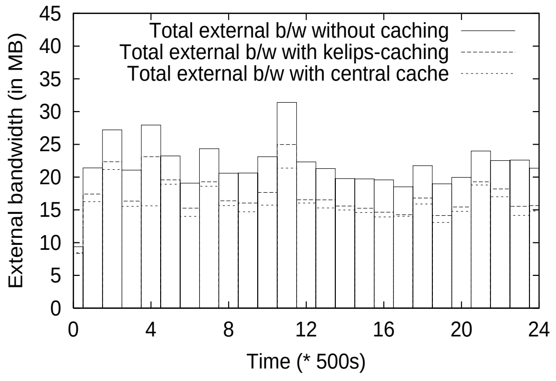

Figure 5: External bandwidth vs Time: Kelips web caching is comparable to that obtained through a central cache.

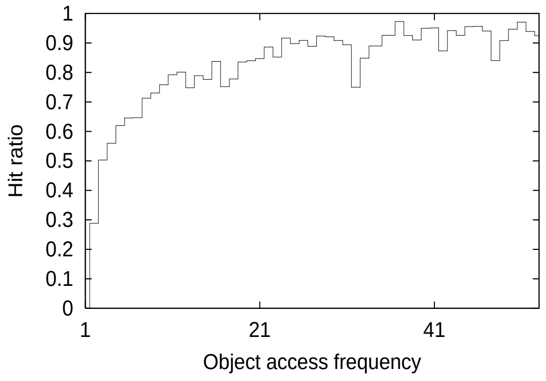

Figure 6: Hit ratio vs Object access frequency: Objects accessed more frequently have higher hit ratios, saturating out at beyond oaf=20.

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 5 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 6). The plot appears to level out beyond a value of oaf=20.

Single Hop (SH) versus Multihop (MH). In the multihop 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

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

affinity group and for at most 3 hops in the target affinity group.

Table 2 compares the hit ratio and average external bandwidth per request for the single hop (SH) and multi-hop (MH) query routing schemes. A comparison with Table 1 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 2 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 5.2.1.

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 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 7 and 8 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 9. 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 9 shows that this is indeed not the case. Kelips web caching thus achieves good load balancing w.r.t. object requests.

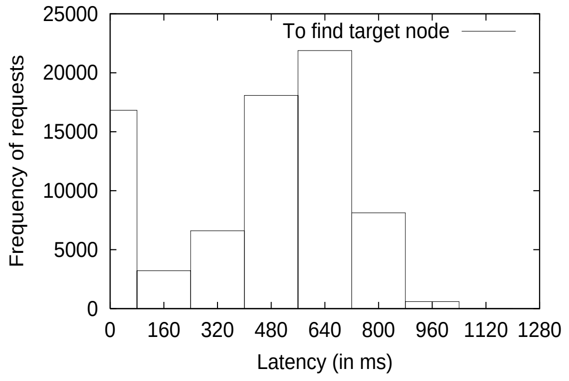

Figure 7: Request frequency vs. Latency to search for a cache copy - SH. Plotted for all requests to the external cache.

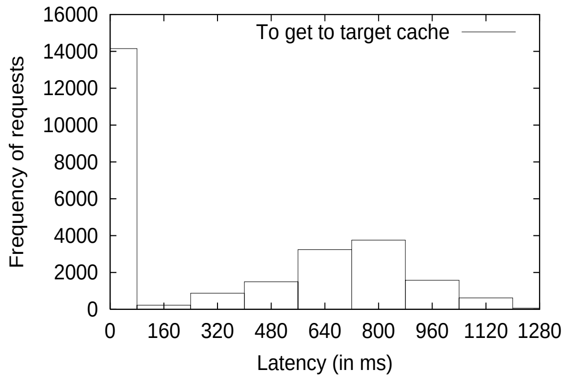

Figure 8: Request frequency vs Latency to access cached copy - SH. Plotted for requests that result in external cache hits.

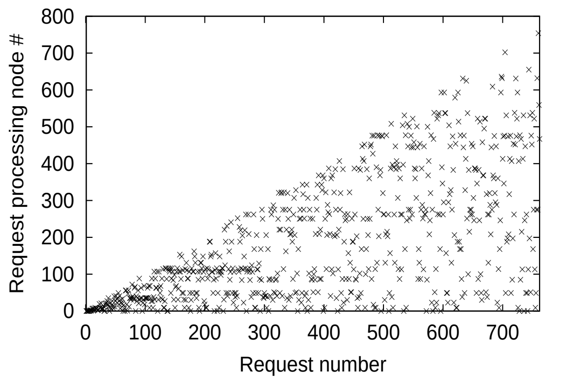

Figure 9: Req processing node# vs Req num (see text in "Load Balancing" for explanation)

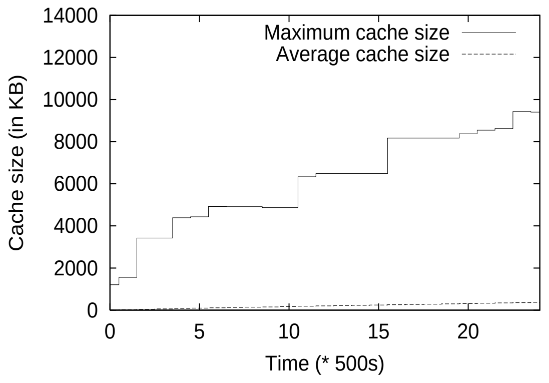

Figure 10: Cache size vs Time: Average and Maximum cache sizes are smaller than 10 MB throughout the trace of duration 12,200 s.

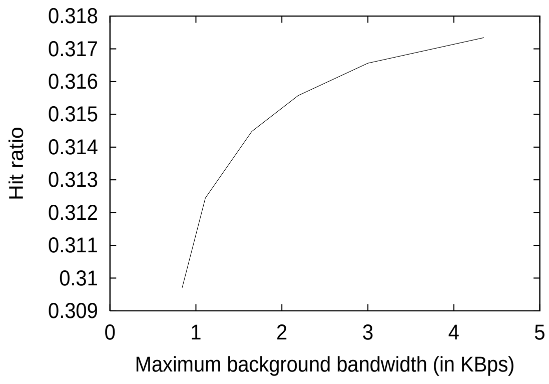

Figure 11: Hit ratio vs Max background bandwidth: Increased background gossip communication cost affects the hit ratio by increasing the number of fresh web object tuples.

Cache Size. Figure 10 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.

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 11 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.

5.2.1 Effect of Churn: Constant Node Arrival and Departure Rates

The experiments in reference [6] 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 [2]. 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 12 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 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 [3].

Figure 13 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.

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

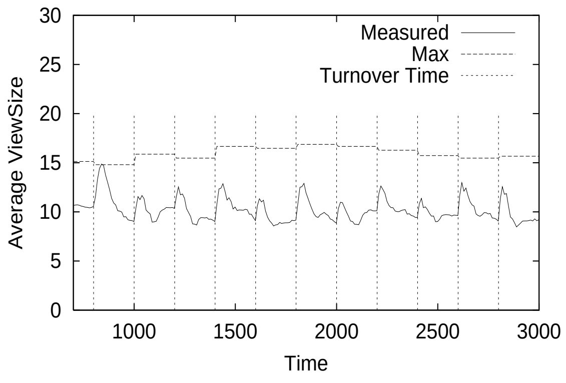

Figure 12: Effect of churn on Affinity Group View Size at a node: Hourly availability traces from the Overnet system are periodically injected into the system (at the times shown by the vertical bars). Churn trace injection epoch for this plot is 200 s.

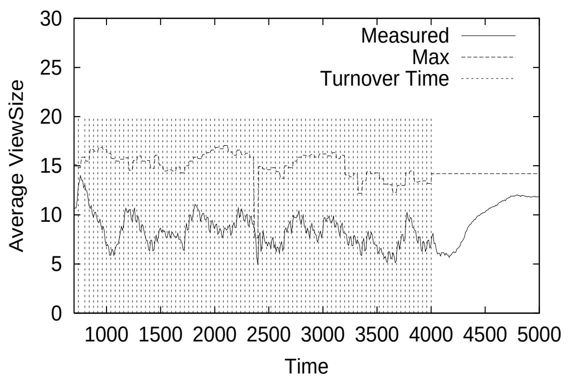

Figure 13: Effect of churn on Affinity Group View Size at a node: Hourly availability traces from the Overnet system are periodically injected into the system (at the times shown by the vertical bars). Churn trace injection epoch for this plot is 40 s.

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 [14]. 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[4]. 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 14 and 15 show the request latency distributions under the effect of churn. A comparison with Figures 7 and 8 respectively, and a glance at Table 2, 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 12, 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 16 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 13), 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.

[4]This is justified by the results of [2] showing that availability characteristics tend to be uncorrelated across clients.

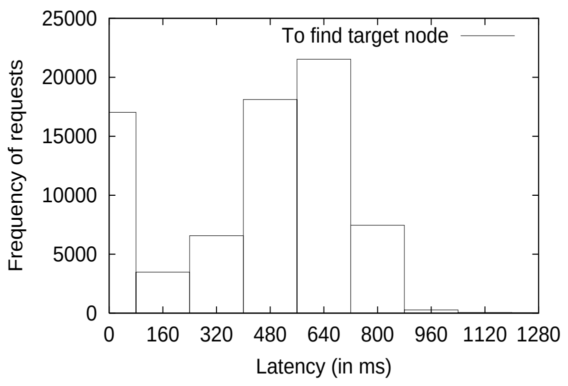

Figure 14: Effect of Churn: Request frequency vs. Latency to search for a cache copy - SH. Plotted for all requests to the external cache. Churn trace injection epoch is 200 s.

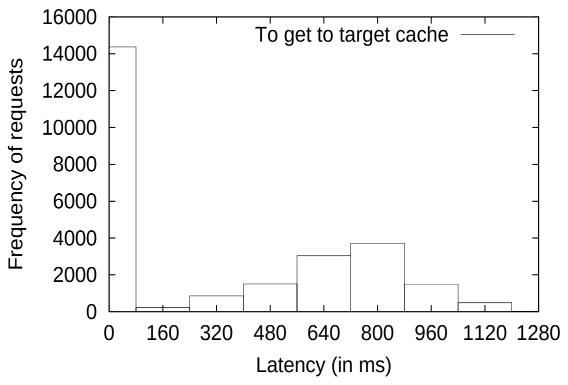

Figure 15: Effect of Churn: Request frequency vs Latency to access cached copy - SH. Plotted for requests that result in external cache hits. Churn trace injection epoch is 200 s.

Note from Figure 8 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 8 (54.4%) are local, we focus on the stability of the non-local p2p cache hits (the "remaining 45.6%"). From Figure 16, 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.

6. SUMMARY

Peer-to-peer applications may be subject to denial of service attacks from extreme stresses with origins typically considered non-malicious. We have studied one such source called churn, that arises from rapid arrival and failure (or departure) of a large number of participants in the system, and we have done so in the context of a peer-to-peer web

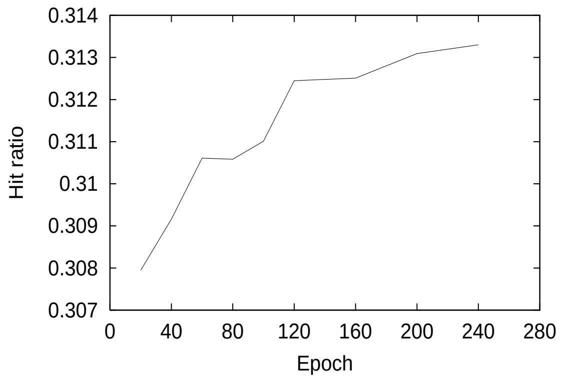

Figure 16: Effect of Churn: Hit rate vs Churn trace injection epoch.

caching application. Other malicious attacks are possible but we do not address these in this paper. This paper has shown how to design a churn-survivable 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. 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. 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.

7. REFERENCES

[1] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, H. M. Levy, "On the scale and performance of cooperative web proxy caching", *Proc. 17th ACM Symposium on Operating Systems Principles*, 1999, pp. 16-31.

[2] R. Bhagwan, S. Savage, G.M. Voelker, "Understanding availability", *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003, pp. 135-140.

[3] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber and M. F. Schwartz, "The Harvest information discovery and access system", *Computer Networks and ISDN Systems*, 28(1-2):119-125, Dec. 1995.

[4] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, K. J. Worrell, "A hierarchical Internet object cache", *Proc. 1996 Usenix Technical Conference*, San Diego, CA, Jan. 1996.

[5] F. Dabek B. Zhao, P. Druschel, J. Kubiatowicz, I. Stoica, "Towards a common API for structured peer-to-peer overlays", *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

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

[7] N. T. J. Bailey, "Epidemic Theory of Infectious Diseases and its Applications", Hafner Press, Second Edition, 1975.

[8] A. Demers, D. H. Greene, J. Hauser, W. Irish, J. Larson, "Epidemic algorithms for replicated database maintenance", *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

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

[10] D. Kempe, J. Kleinberg, A. Demers. "Spatial gossip and resource location protocols", *Proc. 33rd ACM Symposium Theory of Computing (STOC)*, 2001.

[11] D. Liben-Nowell, H. Balakrishnan, D. Karger, "Observations on the Dynamic Evolution of Peer-to-Peer Networks", *Proc. 1st International Workshop Peer-to-Peer Systems (IPTPS)*, 2002.

[12] C. Mohan, "Caching Technologies for Web Applications", Talk at Cornell University, Ithaca, NY, `http://www.almaden.ibm.com/u/mohan`, Nov. 2002.

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

[14] S. Saroiu, P.K. Gummadi, S.D. Gribble, "A measurement study of peer-to-peer file sharing systems", *Proc. Multimedia Computing and Networking (MMCN)*, 2002.

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

[16] J. Wang, "A survey of web caching schemes for the internet", *ACM Computer Communication Review*, 29(5):36-46, Oct. 1999.

[17] D. Wessel, "Squid internet object cache", `http://squid.nlanr.net`.

[18] B.D. Davison, "Web Caching and Content Delivery Resources", `http://www.web-caching.com`

[19] "Modeling Topology of Large Internetworks", `http://www.cc.gatech.edu/projects/gtitm`

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

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

[22] T. Stading, P. Maniatis, M. Baker, "Peer-to-peer caching schemes to address flash crowds", *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.