# Squirrel: A decentralized peer-to-peer web cache

Sitaram Iyer*
Rice University
6100 Main Street, MS-132
Houston, TX 77005, USA
ssiyer@cs.rice.edu

Antony Rowstron
Microsoft Research
7 J J Thomson Close
Cambridge, CB3 0FB, UK
antr@microsoft.com

Peter Druschel
Rice University
6100 Main Street, MS-132
Houston, TX 77005, USA
druschel@cs.rice.edu

## ABSTRACT

*This paper presents a decentralized, peer-to-peer web cache called Squirrel. The key idea is to enable web browsers on desktop machines to share their local caches, to form an efficient and scalable web cache, without the need for dedicated hardware and the associated administrative cost. We propose and evaluate decentralized web caching algorithms for Squirrel, and discover that it exhibits performance comparable to a centralized web cache in terms of hit ratio, bandwidth usage and latency. It also achieves the benefits of decentralization, such as being scalable, self-organizing and resilient to node failures, while imposing low overhead on the participating nodes.*

## 1. INTRODUCTION

Web caching is a widely deployed technique to reduce the latency observed by web browsers, decrease the aggregate bandwidth consumption of an organization's network, and reduce the load incident on web servers on the Internet [5, 11, 22]. Web caches are often deployed on dedicated machines at the boundary of corporate networks, and at Internet service providers. This paper presents an alternative for the former case, in which client desktop machines themselves cooperate in a peer-to-peer fashion to provide the functionality of a web cache. This paper proposes decentralized algorithms for the web caching problem, and evaluates their performance against each other and against a traditional centralized web cache.

The key idea in Squirrel is to facilitate mutual sharing of web objects among client nodes. Currently, web browsers on every node maintain a local cache of web objects recently accessed by the browser. Squirrel enables these nodes to export their local caches to other nodes in the corporate network, thus synthesizing a large shared virtual web cache. Each node then performs both web browsing and web caching.

*This work was done during a summer internship at Microsoft Research, Cambridge.


PODC-21 7/2002 Monterey, California


There is substantial literature in the areas of cooperative web caching [3, 6, 9, 20, 23, 24] and web cache workload characterization [4]. This paper demonstrates how it is possible, desirable and efficient to adopt a peer-to-peer approach to web caching in a corporate LAN type environment, located in a single geographical region. Using trace-based simulation, it shows how most of the functionality and performance of a traditional web cache can be achieved in a completely self-organizing system that needs no extra hardware or administration, and is fault-resilient. The following paragraphs elaborate on these ideas.

The traditional approach of using dedicated hardware for centralized web caching is expensive in terms of infrastructure and administrative costs. Large corporate networks often employ a cluster of machines, which usually has to be overprovisioned to handle peak load bursts. A growth in user population causes scalability issues, leading to a need for hardware upgrades. Another drawback is that a dedicated web cache may represent a single point of failure, capable of denying access to cached web content to all users in the network.

In contrast, a decentralized peer-to-peer web cache like Squirrel pools resources from many desktop machines, and can achieve the functionality and performance of a dedicated web cache without requiring any more hardware than the desktop machines themselves. An increase in the number of these client nodes corresponds to an increase in the amount of shared resources, so Squirrel has the potential to scale automatically.

Squirrel uses a self-organizing, peer-to-peer routing substrate called Pastry as its object location service, to identify and route to nodes that cache copies of a requested object [15]. Squirrel thus has the advantage of requiring almost no administration, compared to conventional cooperative caching schemes. Moreover, Pastry is resilient to concurrent node failures, and so is Squirrel. Upon failure of multiple nodes, Squirrel only has to re-fetch a small fraction of cached objects from the origin web server.

The challenge in designing Squirrel is to actually achieve these benefits in the context of web caching, and to exhibit performance comparable to a centralized web cache in terms of user-perceived latency, hit ratio, and external bandwidth usage. Finally, Squirrel faces a new challenge: nodes in a decentralized cache incur the overhead of having to service each others' requests; this extra load must be kept low. The rest of the paper shows how Squirrel achieves these goals, supported by trace-based simulation results.

Section 2 provides background material on web caching and on Pastry. Section 3 describes the design of Squirrel, and Section 4 presents a detailed simulation study using traces from the Microsoft corporate web caches. Section 5 discusses related work, and Section 6 concludes.

# 2. BACKGROUND

This section provides a brief overview of web caching, and the Pastry peer-to-peer routing and location protocol.

## 2.1 Web caching

Web browsers generate HTTP GET requests for Internet objects like HTML pages, images, etc. These are serviced from the local web browser cache, web cache(s), or the origin web server – depending on which contains a fresh copy of the object. The web browser cache and web caches receive GET requests. For each request, there are three possibilities: the requested object is uncacheable, or there is a cache miss, or the object is found in the cache. In the first two cases the request is forwarded to the next level towards the origin web server. In the last case, the object is tested for freshness (as described below). If fresh, the object is returned; otherwise a *conditional GET (cGET)* request is issued to the next level for validation. There are two basic types of cGET requests: an *If-Modified-Since* request with the timestamp of the last known modification, and an *If-None-Match* request with an *ETag* representing a server-chosen identification (typically a hash) of the object contents. This cGET request can be serviced by either another web cache or the origin server. A web cache that receives a cGET request and does not have a fresh copy of the object forwards the request towards the origin web server. The response contains either the entire object (sometimes with a header specifying that the object is uncacheable), or a *not-modified* message if the cached object is unchanged [11, 21, 22].

Freshness of an object is determined by a web cache using an expiration policy. This is generally based on a time-to-live (ttl) field, either specified by the origin server, or computed by the web cache based on last modification time. The object is declared stale when its ttl expires. HTTP/1.1 allows clients to control the expiration policy for requested objects, such as the degree of acceptable staleness [8]. For the purposes of this paper, we assume the existence of a binary-valued freshness function that computes the freshness of an object based on factors such as time-to-live, time of last modification, time of object fetch from the origin server, and the current time.

## 2.2 Pastry

A number of peer-to-peer routing protocols have been recently proposed, including CAN [13], Chord [19], Tapestry [25] and Pastry [15]. These self-organizing, decentralized systems provide the functionality of a scalable *distributed hash-table*, by reliably mapping a given object key to a unique live node in the network. The systems balance storage and query load, transparently tolerate node failures, and provide efficient routing of queries. Squirrel uses Pastry to store cached Web objects and directory information, and to efficiently locate them. Pastry is described and evaluated in [15]; for continuity and containment, we present a brief description here.

Pastry is a peer-to-peer location and routing substrate that is efficient, scalable, fault resilient, and self organizing. A number of other applications have been built on top of Pastry, including an archival storage utility (PAST) [16] and an event notification system (Scribe) [17]. Pastry assigns random, uniformly distributed *nodeIds* to participating nodes (say $N$ in number), from a circular 128-bit namespace. Similarly, application objects are assigned uniform random objectIds in the same namespace. Objects are then mapped to the live node whose nodeId is numerically closest to the objectId. To support object insertion and lookup, Pastry routes a message towards the live node whose nodeId is numerically closest to a given objectId, within an expected $\lceil log_{2^b} N \rceil$ routing steps. In a network of 10,000 nodes with $b = 4$, an average message would route through three intermediate nodes. Despite the possibility of concurrent failures, eventual message delivery is guaranteed unless $\lfloor l/2 \rfloor$ nodes with adjacent nodeIds fail simultaneously. ($l$ has typical value $8 * log_{16} N$). Node additions and abrupt node failures are efficiently handled, and Pastry invariants are quickly restored.

Pastry also provides applications with a *leaf set*, consisting of $l$ nodes with nodeIds numerically closest to and centered around the local nodeId. Applications can use the leaf set to identify their neighbours in the nodeId space, say for replicating objects onto them.

# 3. SQUIRREL

The target environment for Squirrel consists of 100 to 100,000 nodes (generally desktop machines) in a typical corporate network. We expect Squirrel to operate outside this range, but we do not have workloads to demonstrate it. We assume that all nodes can access the Internet, either directly or through a firewall. Each participating node runs an instance of Squirrel with the same expiration policy, and configures the web browser on that node to use this Squirrel instance as its proxy cache. The browser and Squirrel share a single cache managed by Squirrel; one way to achieve this is by disabling the browser's cache. No other changes to the browser or to external web servers are necessary. This paper focusses on the scenario where Squirrel nodes are in a single geographic region. We then assume that communication latency between any two Squirrel nodes is at least an order of magnitude smaller than the latency to access external servers, and that they are connected by a network of bandwidth at least an order of magnitude more than the external bandwidth.

We now explore the design space for Squirrel, and provide rationale for the combinations of choices adopted. The chief goals are to achieve performance comparable to a dedicated web cache, avail of various potential benefits of decentralization such as scalability and fault-tolerance, and keep the overhead low. We now develop two algorithms for Squirrel: one simple and straightforward, and the other more sophisticated; we compare them in Section 4.

Web browsers issue their requests to the Squirrel proxy running on the same node. If this proxy knows that the object is uncacheable (as discussed in Section 4.1), it forwards the request directly to the origin web server. Otherwise it checks the local cache, just as the browser would normally have done to exploit client-side locality and reuse. If a fresh copy of the object is not found in this cache, then Squirrel essentially tries to locate a copy on some other node. It starts by mapping the object URL (the key) to a node in the network using Pastry, as follows. It computes a SHA-1 [1] hash of the URL to obtain a 128-bit objectId, and invokes the Pastry routing procedure to forward the request to the node with nodeId numerically closest to this objectId. It designates the recipient as the *home node* for this object.

The two Squirrel approaches differ at this point, based on the question of whether the home node actually stores the object, or whether it only maintains a directory of information about a small set of nodes that store the object. (These nodes, called delegates, are nodes whose browsers have recently requested the object, and are likely to have it in their local caches). This bifurcation leads to the *home-store* and the *directory* schemes respectively.

***Home-store:***
In this model, Squirrel stores objects both at client caches and at its home node. The protocol, as depicted in Figure 1, works as follows: if the client cache does not have a fresh copy of the object, it may either have a stale copy or none at all. It correspondingly issues a cGET or a GET request to the home node. If the home node has a fresh copy, it directly connects back to the client and responds with the object or a not-modified message as appropriate.

If the home node instead finds a stale copy in its cache, or if it incurs a cache miss, it issues a cGET or a GET request to the origin server respectively. If the origin server responds with a not-modified message or a cacheable object, then the home node revalidates the local copy or stores the object as appropriate, and forwards a suitable response to the initiating client. On the other hand, if the response is a uncacheable object, then it simply forwards the response to the initiating client. (It does not maintain a negative cache, though that feature may conceivably be retrofitted).

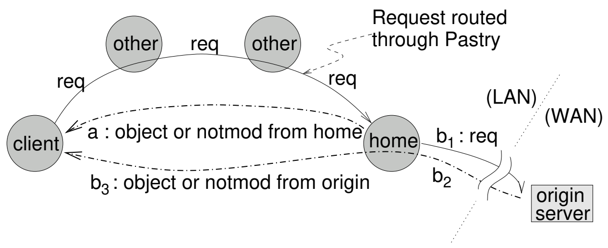



**Figure 1: The home-store scheme (slightly simplified). The request is handled in one of two possible ways, (a) or ($b_1$-$b_2$-$b_3$).**

All external requests to an object are routed through its home node. So this home node normally maintains the most up-to-date copy of the object in the Squirrel network, and does not have to search among other nodes before responding. The exception to this is when the object is evicted from the home node's cache; cache evictions are found to be infrequent even for modest cache sizes, so we keep the design simple and perform an external request in such cases. All objects stored in each node's cache are treated equally by the cache replacement policy, regardless of whether the object was stored there because the node was its home, or because the node accessed it recently (or both).

***Directory:***
This approach is based on the idea that a node that recently accessed an object can be asked to serve it to subsequent clients. In this approach, the home node for an object remembers a small directory of up to $K$ (e.g., 4) pointers to nodes that have most recently accessed the object. The key idea is to redirect subsequent requests to a randomly chosen node from among these (called the delegate), which would be expected to have a copy of the object locally cached. This protocol maintains the invariant that these copies stored at the delegate nodes are the same *version* of the object, as determined by its ETag[1].

This protocol is depicted in Figure 2. Each node maintains a directory for objects homed at that node, along with metadata like the object's ETag, fetch time, last modified time, and any explicit time-to-live and cache-control information. Thus if the home node receives a cGET request for the object, it can apply the expiration policy and validate the request without storing the object itself. Moreover, since the directory points to copies of the same *version* of the object, all these copies go stale independently and simultaneously. If the home node revalidates the object from the origin server at a later time, and if the object is found to be unmodified (matching ETag), then all cached copies with the same version are known to the home node to be valid, and it marks the directory as such.

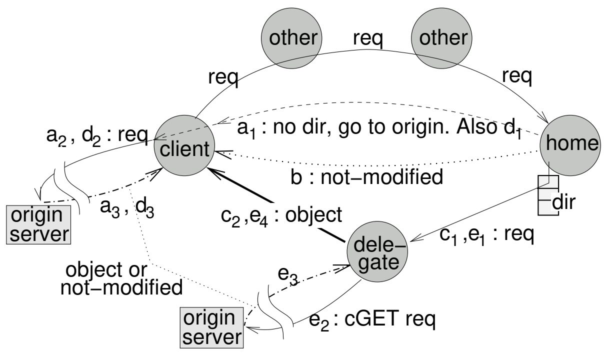



**Figure 2: The directory scheme (slightly simplified). The request is handled in one of five possible ways, (a), (b), (c), (d) or (e).**

In this model, a HTTP request is serviced by Squirrel as follows. As with the home-store model, the cache on the client node is first checked, and a cGET or a GET request is issued to the home node. The home node may never have seen this object before, in which case it has no directory associated with this object (case (a) in Figure 2). It sends a brief message back to the client, and the client makes a request directly to the origin server. The home node treats the client as a tentative delegate for this object, and optimistically adds it to the directory with unknown ETag and metadata. Normally the client soon fetches the object from the origin server, and sends a short message to the home node to update the ETag and metadata. If the object is marked uncacheable, then the client instead sends a suitable message asking the home node to remove the entire directory. (Again, we do not maintain a negative cache).

The second case is when there is a non-empty directory at the home node, and is known to point to nodes with fresh

[1]If an object fetched from the origin server does not have an ETag, then Squirrel synthesizes an ETag for internal use by hashing the object contents.

copies of the object. If the client's request was a cGET, and if the ETags match, then the request can be validated immediately and a not-modified reply sent to the client (case (b)). Otherwise the request is forwarded to a randomly chosen delegate from the directory, and the client is optimistically added to the directory. The delegate receives the request, which the home node has augmented with directory freshness and lifetime information. Therefore, if the directory is fresh and the delegate considers its copy of the object to be stale, it is told to update its information so the object is made fresh. (This can happen if another delegate has accessed the object and revalidated the directory in the interim). The delegate then directly returns the requested object to the client (case (c)).

The final pair of cases arises when the directory is populated with pointers to nodes that have stale copies of the object. If the client's request was a cGET (where the client may or may not be a delegate), then a brief message is sent to the client to perform its own external request (case (d)). The client later sends an update message for the metadata, whereupon the home node either revalidates its directory, or replaces all entries by this client if the new version is different.

Instead, if the client's request was a GET, then the home node forwards it to a randomly chosen delegate from the directory (so that the entire object need not be fetched from the origin server if unmodified). To prevent multiple delegates from making simultaneous external connections, the home node temporarily removes all other entries from its directory. The delegate makes an external cGET request to validate its copy (case (e)), and sends the object to the initiating client. The delegate also sends a message to the home node, either to reinstantiate all its earlier entries (if the delegate got a not-modified response to its cGET), or to update the metadata and set the client and the delegate as the only two entries. Thus, ($e_2$) in the figure can only be a cGET request, and ($e_4$) can only be the object.

A special case happens when a request reaching a delegate finds the object missing from its local cache, despite the home node having a pointer to it. This is observed to be an infrequent occurrence (even with small per-node caches), and has two possible reasons: the delegate may have either evicted the object from its cache, or may have requested it only very recently (so the current request is due to optimistic addition of the delegate to the home node's directory). In the first case, the delegate informs the home node to remove it from the directory for the object, and the home forwards the request to another delegate from the directory, if any exist. In the second case, the delegate keeps the forwarded request pending until the first one completes.

***Node arrival, departure and failure:***

Both schemes need to cope with frequent variations in node membership. The underlying Pastry network ensures reliable routing (even in the presence of many concurrent failures), but Squirrel needs to perform the following extra operations.

When a node joins the Squirrel network, it automatically becomes the home for some objects, but does not store the objects or directories yet. If no information is passed to the new node, then Squirrel would still continue to function correctly; but the small fraction of requests destined for this new node would be routed to the origin server, resulting in a small performance reduction. Squirrel overcomes this performance drop by leveraging Pastry's feature of informing the peer-to-peer application of changes in a node's leaf set[2]. When this occurs, the two neighboring nodes in the nodeId space transfer the necessary objects or directories to the newly joined node. The directory approach has less state to transfer compared to the home-store approach, which has to transfer the actual objects. If this state is too large, then the neighbors can sort objects by frequency of access, and transfer the most frequently accessed ones. Given the Zipf-like distributions of many web workloads, this optimization allows a lightweight joining protocol.

Failure of a Squirrel node will result in a corresponding fraction of objects and directories abruptly disappearing from Squirrel. Future requests for the object will be routed to the node that has now become numerically closest to the objectId. In the home-store model, this node queries the origin server for subsequent requests. In the directory model, this node does not try to reconstruct the lost directory; instead, it simply creates a new directory, and asks the client to query the origin server. This only results in transitory and graceful performance reduction in both cases even when many nodes fail; the extent of damage is lesser in home-store than in directory. Finally, nodes that are capable of announcing their desire to leave the system can transfer some of their directories or objects to their two immediate neighbors in nodeId space.

In the directory model, a delegate node may fail without notice; when a home node chooses this delegate and immediately discovers its inability to connect to it, it removes this delegate from all its directories, and forwards the request to another delegate if available.

***Comparison:***

The two approaches represent extremes of the design space, based on the choice of storage location. Despite their differences, to a first level of approximation, they both achieve comparable performance to each other and to a centralized web cache. Their hit ratios are comparable if the cache expiration and replacement policies are the same, and if the caches are sufficiently large. User-perceived latency is dominated by requests that need to be serviced from the origin server; comparable hit ratios implies comparable user-perceived latency, and minor differences in the number of LAN hops are imperceptible. Section 4 evaluates these factors more thoroughly, and analyzes second-order effects due to cache sizes, etc.

In terms of the overhead imposed on individual nodes, and on the degree of resilience to node failures, the two schemes are surprisingly different. The more sophisticated directory scheme may initially appear conducive to better load balancing, since popular objects are associated with rapidly changing directories of recently accessing clients, and subsequent load is dispersed onto them. However, in practice, it is the simpler and more elegant home-store scheme that achieves drastically lower load. It capitalizes on a hash function to distribute requests evenly onto home nodes. This natural load balancing results in low peak and sustained overhead, and graceful performance degradation on node failures. Section 4.4 evaluates and explains this result.

[2]The leaf set contains the $l$ nodes with the numerically closest nodeIds to node ($l/2$ smaller and $l/2$ larger).

# 4. EVALUATION

This section evaluates the home-store and the directory designs against each other, and against a dedicated web cache. These approaches are compared on three main metric types: performance (latency, external bandwidth and hit ratio), overhead (load and storage per machine), and fault tolerance.

## 4.1 Trace characteristics

Web caching is a domain where the merits of a solution are largely determined by its performance on a variety of real workloads. We evaluate Squirrel by trace-driven simulation on two substantially different proxy cache access logs on the boundaries of corporate intranets. The *Redmond* trace from Microsoft Corporation in Redmond contains over 36,000 active clients in one geographic location, issuing a total of 16.4 million requests in a single day. The *Cambridge* trace from the Microsoft Research Lab in Cambridge contains 0.97 million requests issued by 105 clients over 31 days.

These Microsoft ISA proxy server access logs reflect little of the object cacheability information that it obtains from requests and responses. In practice, different proxy servers decide whether an object is cacheable based on some combination of checks for cache-control headers, existence of cookies (in HTTP/1.0), password protection, '/cgi-bin/' in the URL, etc [5, 21, 24]. For simulation purposes, we use the available information to approximately deduce cacheability; our main goal is to derive an understanding of Squirrel behavior on diverse workloads, rather than make a case for web caching. Thus we define *static objects* as those accessed with a HTTP GET request (without SSL), without CGI-like characters such as '?', '=', or the word 'cgi' in the URL. The following simulations treat all and only these as cacheable, so all *dynamic requests* are immediately forwarded by Squirrel on the client node to the origin server. The following table presents some trace characteristics.

| | Redmond | Cambridge |
|---|---|---|
| Trace date | May '99 | July–Aug '01 |
| Total duration | 1 day | 31 days |
| Number of HTTP requests | 16.41 million | 0.971 million |
| Mean request rate | 190 req/s | 0.362 req/s |
| Peak request rate | 606 req/s | 186 req/s |
| Number of objects | 5.13 million | 0.469 million |
| Total object size | 49.3 GB | 3.37 GB |
| Number of clients | 36782 | 105 |
| Mean req/client/hr | 18.5 | 12.42 |
| Number of static objects | 2.56 million | 0.226 million |
| Static object requests | 13.84 million | 0.727 million |
| Mean static object reuse | 5.4 times | 3.22 times |
| Total static object size | 35.1 GB | 2.21 GB |
| Total external bandwidth | 88.1 GB | 5.7 GB |
| Hit ratio | 29% | 38% |

Mean and peak request rate values represent the load incident on a centralized proxy server, and motivate the need for dedicated, overprovisioned machines for a centralized cache. The mean requests/client/hour signifies users' average browsing rate (including web pages, images, etc). Mean reuse of static objects is the average number of times each static object is accessed, and denotes the potential benefit of web caching. Total external bandwidth usage is directly obtained from the trace. It differs from the total object size due to request and not-modified messages, and also because objects need to be fetched again when they are modified at the origin server or evicted from the centralized cache. Finally, *hit ratio* is defined as the fraction of all objects (static and dynamic) that get satisfied from the web cache. These numbers (29% and 38%, derived using the status field in the log) are found to roughly compare with various web cache hit ratios reported in the literature [5, 24].

For both traces, we create simulated networks consisting of one node for each web client from the trace; all of these act as Squirrel proxy nodes. At initialization, the Squirrel cache is warmed by inferring freshness of objects as follows. On the first request for each object in the trace, if the centralized web cache has returned a valid copy or a not-modified message, then we warm the Squirrel home node or a randomly chosen delegate (depending on the approach) with this object. In the not-modified case, we also warm the cache at the requesting client with a stale copy. During the experiment, the nodes issue requests from the log at moments in time specified therein. We use the log status field to infer the expiration policy decision of the centralized cache for the requested object (so that the simulation does not have to adopt a separate expiration policy), and also whether the object was found to be modified at the origin server. For the centralized cache numbers below, we use the same inferred values for fair comparison.

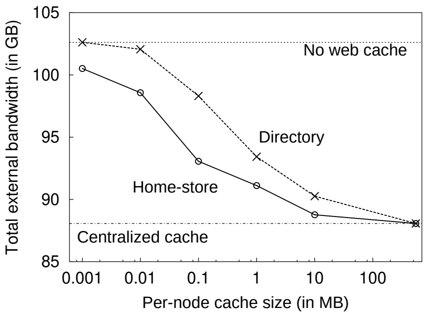
## 4.2 External bandwidth and hit ratio

We define external bandwidth as the number of bytes transferred between Squirrel and the origin servers. In this section, we compare external bandwidth savings achieved by the two Squirrel algorithms, and measure the impact of limiting the per-node cache size. We need to know the size of each message to compute external bandwidth traces, so we assume values from the following table.

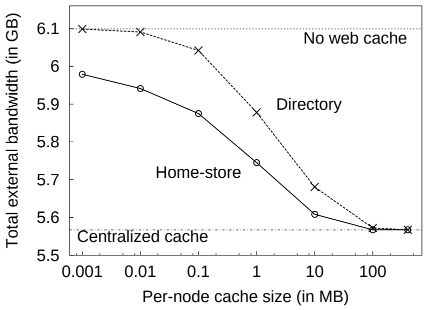
| | |
|---|---|
| Request headers | 350 bytes (estimate) |
| Object transfer and response headers | Response-size field in the proxy log |
| Not-modified responses | 150 bytes (estimate) |

Figure 3 shows the external bandwidth in GB over the entire period of each trace. The local cache on each node is limited to different sizes (as depicted on the horizontal axis) by using an LRU cache replacement policy. The two dotted horizontal lines represent the external bandwidth if no cache is used, and if a centralized cache with a sufficiently large disk storage capacity is used. These two are calculated by projecting the inferred expiration policy, the above message size estimates, and the object cacheability estimate on the centralized cache model with infinite storage. (Observe that these centralized cache bandwidths roughly tally with those measured from the trace, thus supporting the validity of our inferences). The difference between the two values denotes the maximum external bandwidth reduction that web caching obtains; as expected, the greater degree of sharing in the Redmond trace implies a larger saving.

First consider only the home-store curves in both graphs. Note that a reasonable amount of disk space, e.g. 100MB, donated to Squirrel by every node proves sufficient to lower external bandwidth consumption to that close to a dedicated web cache. The fact that this is true for both traces speaks strongly to the effective utilization of per-node caches in the Squirrel network. The horizontal axis is logarithmic, so external bandwidth gradually rises if this per-node contribution is drastically decreased. This indicates that even a very small amount of cache space donated by each node aggregates across nodes to achieve a significant benefit.




*(a) Redmond*




*(b) Cambridge*

**Figure 3: Total external bandwidth usage as a function of per-node cache size. Note that lower values are better, and that Y-axes start from non-zero positions.**

Next consider the difference between the home-store and the directory designs. The home-store approach uses a hash function to uniformly distribute objects across nodes. The directory scheme, on the other hand, only stores objects in the respective client caches. To maintain the cache size limit, a node that rapidly browses many objects will have to evict fairly recently accessed objects from its cache. This results in an imbalance, where the heavily browsing nodes are forced to evict fairly recently accessed objects, while less active nodes may cache much less recently accessed objects. Cache evictions from the heavily browsing nodes therefore lead to increased cache misses and higher external bandwidth use, as compared to the home-store approach.

As quantified in the next paragraph, the home-store approach requires more per-node and total storage than the directory approach, due to copies stored both at clients and at home nodes. Despite this extra storage requirement, it is interesting to observe that the home-store scheme makes overall more effective use of the available cache storage, as indicated by its lower external bandwidth for a given per-node cache size.

The table below describes the maximum possible storage used by the two schemes for the Redmond trace, and is measured by letting all nodes potentially contribute infinite storage. Note that the difference between total storage for the two schemes (i.e., 35989MB = 35.1GB) is equal to the total static object size from the trace characteristics table; also note that a dedicated web cache with this much disk space can store the entire content. Secondly, although the maximum storage on some nodes is relatively high (1.6GB), we have shown that limiting it to say 100MB only causes a slight increase in external bandwidth.

| (Redmond) | Home-store | Directory |
|---|---|---|
| Total | 97641 MB | 61652 MB |
| Mean per-node | 2.6 MB | 1.6 MB |
| Maximum per-node | 1664 MB | 1664 MB |

Hit ratio of the Squirrel cache is indirectly related to external bandwidth. With increasing per-node contributions, the hit ratio approaches that of a centralized cache with enough storage. At about 100MB per node, Squirrel achieves 28% and 37% for Redmond and Cambridge traces (with either model), consistent with the trace characteristics in Section 4.1.

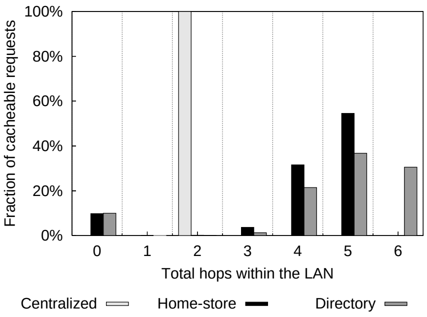
## 4.3 Latency

User-perceived latency is the time between issuing a request and receiving a response back at the client. We assume that communication latencies within the LAN are of the order of a few milliseconds, and are at least an order of magnitude smaller than external latency (say across or between continents). Also, request processing at a node (excluding transfer time) rarely takes more than a few milliseconds. Since hit ratio for Squirrel is comparable to that of a centralized cache, the user-perceived latency per request will approximately be the same, even though Squirrel makes a small number of LAN hops. For large objects, the higher LAN bandwidth implies that the reduction in overall latency is considerable. The purpose of the following experiment is to measure the number of application-level hops for requests within the Squirrel cache, and show that this number is small. By the side, we will also make some observations about forwarding behavior of the directory model.

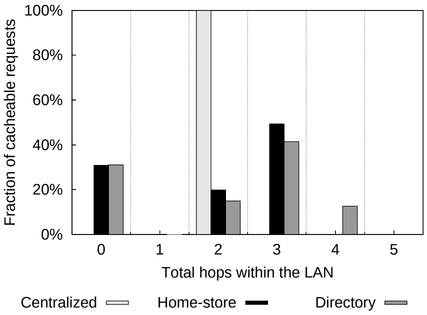
Consider Figure 4. A centralized web cache always incurs two hops within the LAN – request and response. Consider the Redmond case; most requests in the home-store model take between four and five hops, since for Pastry, $4 < \log_{16}(N = 36782) + 1 < 5$. The directory model incurs an extra hop per request that is actually forwarded. The difference between the mean number of hops for the two schemes is $4.56 - 4.11 = 0.45$, and is smaller than 1; therefore more than half the requests in the directory scheme are handled by the home node itself, and do not have to get redirected. These are when the parent responds with short messages like 'go to the origin server' in cases (a) or (d), or 'not-modified' in case (b) of Figure 2.

About 10% of requests for the Redmond trace, and about 28% of requests for the Cambridge trace are serviced from the local node[3]. This is due to two reasons. First, some cGET queries issued by the browser (usually IE) actually get satisfied by the centralized web cache (ISA), due to a difference between their default cache expiration policies. Secondly, in the home-store model, a small fraction of requests (1% for Cambridge) is due to client nodes that are coincidentally also home nodes for the requested object.

[3]The traces exclude information about local browser cache hits, so these are requests that were originally satisfied by the centralized web cache that are now being satisfied by Squirrel on the local node.




*(a) Redmond: Mean 4.11 hops for home-store and 4.56 hops for directory.*




*(b) Cambridge: Mean 1.8 hops for home-store and 2.0 hops for directory.*

**Figure 4: Number of LAN hops per request satisfied by Squirrel.**

In intranets with higher communication latency, multiple hops may become significant. In such environments, a simple optimization called *route-caching* can be used to generally reduce the number of hops. In essence, each node can maintain a loosely consistent cache of nodeId to IP address mappings of other nodes in the network. Subsequent requests dispatched to the same node can use this cache to connect directly and bypass Pastry routing. Route-caching is not used in the above experiments, but a separate test is seen to yield 98% and 48% hit rates on the route cache for the Cambridge and Redmond workloads.

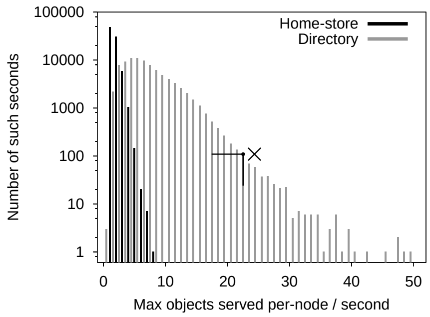
To summarize, the latency induced by Squirrel is small on a cache hit, and is otherwise overshadowed by latency to the origin server.

## 4.4 Load on each node

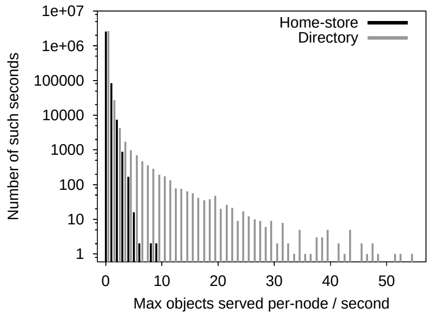
We measure the extra overhead on a node as a result of its participation in Squirrel, in terms of the number of objects served or validated, the number of bytes transferred, and the number of connections processed. To capture the essence of bursty versus sustained load, we separately measure incident load per second and per minute. Frequency plots for the maximum number of objects served by any node on these two timescales are shown in Figures 5 and 6. For presentational clarity, the horizontal axes in some graphs are sampled by displaying only every second or fourth bar. Care was taken not to remove any spikes as a result of this sampling.

Figure 5 indicates that nodes in the directory model incur bursts of relatively high load, up to 48 and 55 objects/s in the two traces. Moreover, in an average second, there is some node in the network that has to serve as many as 6.6 requests in the Redmond case. In contrast, nodes in the home-store model are never observed to serve more than 8 and 9 objects per second, for the two traces.

While the objects/second metric depicts bursts of load, measurements of objects/minute (in Figure 6) represent more sustained periods of load. Again, there is a certain minute during the course of the experiments, when some node in the directory model serves as many as 388 and 125 objects (for the two traces). The home-store model produces a far lower load on the nodes, and incurs a peak load of only 65 and 35 objects (for the two traces) served by some node on two occasions. Moreover, for home-store, the similarity between the maximum load numbers for the two traces, both on per-second and per-minute granularities speaks to the scalability of the system.




*(a) Redmond: Average maximum load for a node is 1.5 objects/s and 6.6 objects/s for home-store and directory respectively.*

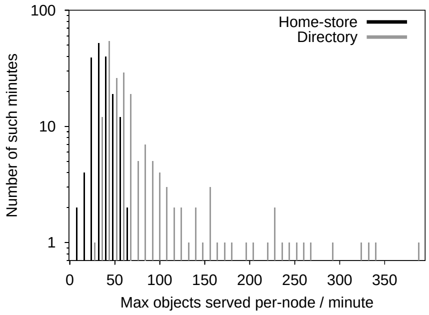



*(b) Cambridge: Average maximum load for a node is 0.038 objects/s and 0.027 objects/s for home-store and directory respectively.*

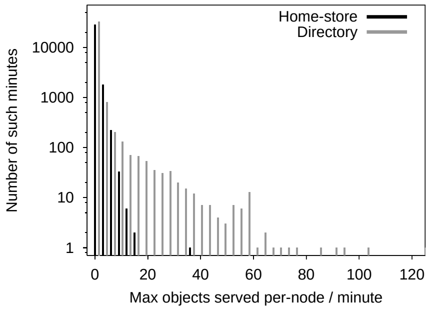
**Figure 5: Load, in maximum objects served per *second*. E.g., the point marked $\times$ in (a) says that there are a hundred individual seconds over the day during which some node services 23 requests. Note that Y-axes are on a logarithmic scale.**

All the above constitute the maximum loads incurred in any given second or minute. However, the *average* load on any node during any second or minute (not shown in graph captions) is extremely low, at 0.31 objects/minute for Redmond with both models. This indicates that Squirrel performs the task of web caching with a negligible fraction of total system resources.

It is interesting to observe why the home-store model causes drastically lower load on machines compared to the directory model. It may seem as though the opposite were true. In the home-store model, the home node for a popular web object may be compelled to serve the object to many simultaneously accessing clients, and is vulnerable to high load. In the directory scheme, the load to serve an object gets distributed among the most recently accessing delegate nodes from the directory. For a popular object, these nodes keep changing rapidly. Hence one might expect the directory approach to be superior; however, in practice, this contrary proves true, for the following reason.




*(a) Redmond: Average maximum load for a node is 36 objects/minute and 60 objects/minute for home-store and directory respectively.*




*(b) Cambridge: Average maximum load for a node is 1.13 objects/minute and 0.7 objects/minute for home-store and directory respectively.*

**Figure 6: Load, in maximum objects served per *minute*. Note that Y-axes are on a logarithmic scale.**

There is a drawback with the directory approach, viz. it implicitly relies on access patterns to distribute load, whereas the home-store scheme simply uses a hash function. Consider, in the directory scheme, a node Q that accesses a web page with many embedded images. Then home nodes corresponding to each of these images add Q to their respective directories. If another client subsequently requests the same web page, then it will simultaneously issue requests for a large number of images. These requests are routed to a widely distributed set of home nodes, but all get forwarded to delegate Q, and cause a burst of substantially high load on Q. Similarly, a heavily browsing client may traverse several rich webpages, and may induce many home nodes to point to it. Later, another client may access the same set of related documents and subject this node to periods of heavy load. There is implicit evidence for this: consider the 388 objects/minute maximum; this is likely to be the result of heavy browsing than a web page with many images. The home-store approach, in contrast, does not have this bursty incident load problem, due to the low probability that many popular objects map via the hash function to a single home node.

In summary, the home-store model shows far less overhead than the directory model on two very different workloads. We have performed analogous experiments on two other publicly available traces, and with other metrics such as bandwidth and number of incident and forwarded connections, and have obtained similar results.

## 4.5 Fault tolerance

This section evaluates the resilience to failure of Squirrel and centralized caches. We distinguish between *connectivity loss* and *performance degradation* due to unavailability of cached content. Network connectivity for individual nodes is independent of web caching, so this section focusses on the latter aspect.

Both Squirrel and a centralized web cache can lose connectivity to the Internet due to a link, router or firewall failure. In this event, they both continue to serve cached content to their clients. A second failure mode is when an internal router or link fails, partitioning the nodes into two sets. The centralized cache would provide cached content only to nodes in its own partition. The Squirrel peer-to-peer network would automatically reorganize itself into two separate systems, if the partitions are sufficiently large. Each would provide its clients access to a fraction of the cached content.

The third, and more prevalent scenario is when connectivity exists to the Internet, but web cache nodes individually fail. If a centralized cache is composed of a single machine, then its failure would imply complete loss of cached content. In cluster-based web caches, there is a graceful, but significant degradation with the failure of each node. In contrast, Squirrel nodes, usually being desktop machines, are expected to fail, or be rebooted or be switched off at night regularly [2]. As seen in Section 3, unannounced node failures result in losing a fraction of the cached content, so Squirrel's overall performance degrades gracefully with the number of failed nodes.

We analyze the impact of a single node failure on the hit ratios observed by other nodes. Let $N$ and $O$ be the number of nodes and (static) objects, $\bar{H}$ and $H_{max}$ be the per-node mean and maximum number of objects homed at that node, and $\bar{S}$ and $S_{max}$ be the per-node mean and maximum number of objects that are the sole entries from their home node's directories. Then the average and worst-case impact of one failed node in the home-store model are $\bar{H}/O$ $(= 1/N)$ and $H_{max}/O$, whereas those for the directory model are $(\bar{H} + \bar{S})/O$ and $\max\{H_{max}, S_{max}\}/O$ respectively.

Measurements for the Cambridge trace give us average and worst-case system performance reductions of 0.95% and 3.34% per failed node in the home-store scheme, compared to 1.68% and 12.4% in the directory scheme. The directory scheme thus proves slightly more vulnerable in the average case when either the home node or the delegate fails, but in the worst case, a node storing say a large web page with many embedded images can fail and lose considerable data. The corresponding numbers for the Redmond trace are 0.0027% and 0.0048% for home-store, and 0.198% and 1.50% for the directory scheme. Thus the directory scheme is somewhat more vulnerable to node failures, especially in the pathological case. The average impact due to multiple node failures can be estimated by linearly scaling the average impact numbers for single node failures.

Machines being rebooted have little effect on Squirrel performance, as they are back online within a few minutes (assuming the objects/directories being cached on the machine are stored on disk). The Farsite study [2] of the Microsoft Corporation internal network shows that about 15% of the machines are likely to be switched off each night or weekend; this means that on average 15% of the cached content in the home-store approach may be lost. However, in reality the impact may be much less, because most machines are shut down gracefully, so the most popular cached content on a node can be transferred to the new home nodes.

### 4.6 Discussion

There are two secondary effects that have the potential to further increase the benefits due to decentralized web caching. Firstly, web workloads often have poor locality, so web caches incur substantial disk traffic. Disk throughput may limit the performance of a centralized web cache to be disk bound, whereas distributing the data across many nodes allows many disks to fetch data in parallel, improving throughput. Secondly, Squirrel pools main memory from many participating nodes. Figure 3 suggests that even a small cache size (such as 10MB) on each node suffices to capture most cache hits. This pooled memory may allow most cache hits to be serviced from memory rather than from disk, whereas a centralized cache is less likely to have sufficient main memory for this to happen. Our simulation does not consider these two factors.

While evaluating the home-store model, the two trace-based workloads we studied achieve low overhead and good load balancing among the participating nodes. This suggests that the natural load balancing achieved in Squirrel through its hash function works well in practice. Yet, workloads with highly popular objects can in theory overload a home node. To address this case, Squirrel maintains a threshold of maximum acceptable rate of requests incident on each node per document. If this limit is exceeded, the hotspot node sheds load by replicating the object on the previous node along the Pastry route from a client. The locality properties of the Pastry overlay network ensure that each of these additional cached copies absorbs a significant fraction of client requests, and that these copies are placed in proximity of interested clients. This technique is proposed and evaluated in [16]; it dynamically adapts the number of cached copies to match demand, and is found to effectively disperse load arising from popular objects.

The benefits of web caches are significant enough to warrant their extensive deployment across the Internet. We believe that a cheap, low-management and fault resilient solution like Squirrel might be the preferred alternative for web caching in large intranets. Furthermore, the experiments in this section let us conclude that under our assumptions, the simpler home-store algorithm is the superior choice.

However, under different sets of assumptions, like geographically distributed Squirrel networks with higher internal latency, some design choices in Squirrel may have to be reevaluated; in fact, the directory approach may prove to be more attractive. We plan to study such scenarios in future work.

## 5. RELATED WORK

Squirrel can be seen as a combination of cooperative web caching, where caches share Internet objects among themselves [3, 6, 9, 20, 23, 24], and peer-to-peer request routing, characterized as decentralized, self-organizing, and fault-resilient [13, 15, 19, 25].

Cooperative web caching (where autonomous web caches coordinate and share content) is found to be useful in improving hit ratio in a group of small organizations. There are many forms, viz. (1) a hierarchical web cache, where upper layers of the hierarchy often cover geographically larger domains [6], (2) hash-based schemes, where clients hash a request to one of multiple dedicated caches, (3) directory-based schemes that maintain a centralized table of references for redirecting every client request, and (4) multicast-based schemes [22].

In view of the problems arising due to weak integration between separate cache nodes in the above methods, a *Distributed WWW cache* has been proposed, with an approach analogous to the xFS file system. A cluster of dedicated web proxy machines is organized into a hierarchy, with frontends, backends and managers. These coordinate to achieve explicit hotspot elimination, fault tolerance and scalability [10].

Squirrel adopts a different paradigm from all the above, in requiring no more infrastructure than the client browser caches themselves. In terms of object location, Squirrel is a peer-to-peer routing-based scheme. It can be viewed as a hash-based caching scheme that is self-organizing and fully decentralized. The directory model for Squirrel ties in with directory-based cooperative caching, but instead of a single directory, Squirrel decentralizes the directories for each object onto randomly distributed home nodes. Just as in the Distributed WWW cache, Squirrel achieves fault tolerance and scalability, and performs sufficient load balancing to avoid hotspots.

MangoSoft's CacheLink product [14] supports sharing of web browser caches, as in Squirrel. The details of how CacheLink achieves this and its general performance are not available. Currently, 250 is the maximum number of machines supported by CacheLink; in comparison, Squirrel scales to many tens of thousands of nodes.

Several systems have recently been proposed to address the problem of flash crowds [12, 18]. Like Squirrel, they use the peer-to-peer paradigm to leverage pooled resources to improve web performance. Unlike Squirrel, they focus on the complementary problem of shielding servers from load spikes resulting from flash crowds.

Distributed storage systems come in many flavors, like persistent peer-to-peer stores such as PAST [16] and CFS [7], distributed filesystems, distributed shared memory systems, etc. with different objectives and tradeoffs. From a conceptual standpoint, Squirrel explores a combination of tradeoffs associated with the closely related class of peer-to-peer applications that maintain soft state and desire low latency of access. Ideas from Squirrel may prove pertinent in the design of similar systems, particularly in a peer-to-peer context.

## 6. CONCLUSION

This paper designs and evaluates a decentralized, peer-to-peer web cache called Squirrel, and shows it to be feasible, efficient, and comparable in performance to a dedicated web cache in terms of latency, external bandwidth and hit ratio. At the same time, Squirrel has the advantages of being inexpensive, highly scalable, resilient to node failures and of requiring little administration. Squirrel imposes a small extra load on each machine, but for the proposed design, this load is shown to be low on a range of real workloads. Finally, given a peer-to-peer routing substrate, it is very easy to deploy Squirrel in a corporate network.

## Acknowledgments

We thank Miguel Castro, Indranil Gupta, Andrew Herbert, Anne-Marie Kermarrec, Animesh Nandi, Juan Navarro, Santashil Palchaudhuri, Marc Shapiro, Atul Singh, and the anonymous reviewers for their comments.

## 7. REFERENCES

[1] F. 180-1. Secure hash standard. Technical Report Publication 180-1, Federal Information Processing Standard (FIPS), National Institute of Standards and Technology, US Department of Commerce, Washington D.C., Apr. 1995.

[2] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of ACM SIGMETRICS*, pages 34–43, June 2000.

[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] M. Busari and C. Williamson. On the sensitivity of web proxy cache performance to workload characteristics. In *Proceedings of IEEE INFOCOM*, Anchorage, Alaska, Apr. 2001.

[5] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: the devil is in the details. In *Proceedings of the Workshop on Internet Server Performance*, Madison, WI, May 1999.

[6] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 Usenix Technical Conference*, San Diego, CA, Jan. 1996.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[8] A. Dingle and T. Partl. Web cache coherence. In *Proceedings of the fifth International World Wide Web Conference*, Paris, May 1996.

[9] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy Squid. In *Proceedings of the Workshop on Internet Server Performance*, Madison, WI, June 1998.

[10] M. Kurcewicz, W. Sylwestrzak, and A. Wierzbicki. A distributed WWW cache. *Computer Networks And ISDN Systems*, 30(22-23):2261–2267, Nov. 1998.

[11] M. Nottingham. Caching tutorial for web authors and webmasters. http://www.mnot.net/cache_docs/.

[12] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *1st International Peer To Peer Systems Workshop (IPTPS 2002)*, Cambridge, MA, Mar. 2002.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[14] P. Romine. LAN-Based web caching for accelerated web access. Mangosoft Technical White Paper, http://www.mangosoft.com/products/cachelink.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001.

[16] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[17] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the third International Workshop on Networked Group Communications*, London, UK, Nov. 2001.

[18] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *1st International Peer To Peer Systems Workshop (IPTPS 2002)*, Cambridge, MA, Mar. 2002.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, San Diego, California, Aug. 2001.

[20] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report CS98-04, Department of Computer Science, University of Texas at Austin, May 1998.

[21] References for cacheability of web content. http://www.cs.rice.edu/~ssiyer/r/squirrel/links.html.

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

[23] D. Wessel. Squid internet object cache. http://squid.nlanr.net/.

[24] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, Dec. 1999.

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

*http://www.cs.rice.edu/~ssiyer/r/squirrel/*