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

Paper by Sitaram Iyer, Antony Rowstron, Peter Druschel

Write-up by Alexander Prohaska

# Contents

# Chapter 1

# Introduction

Squirrel[6] is a decentralized peer-to-peer web cache. It is scalable, self-organizing and resilient to peer failures. Without the need for additional hardware or administration it is able to achieve the functionality and the performance of a traditional centralized web cache. It is proposed to run in a corporate LAN type environment, located e.g. in a building, a single geographical region.

Squirrel is build up on Pastry[9], an object location and routing protocol for large-scale peer-to-peer systems, which provides the mentioned features.

The goals and the motivation for web caching are decline of load on external web servers, corporate routers and of course external traffic (traffic between the corporate LAN and the internet) which is expensive, especially for large organizations.

Squirrel is a possibility to achieve these goals without the use of a centralized web cache or even clusters of web caches.

# Chapter 2

# Background

## 2.1 Web caching

When a user wants to view an internet object (html page, picture or whatever) the browser starts a request for this object. The browser first looks in the local browser cache for the requested object. If it's not there (cache miss) the browser forwards the request towards the next step to the origin web server which hosts the object of interest. If the object is in the cache the browser checks if it is fresh[1]. If it's not fresh, e.g. the time-to-live is passed by, the browser issues a conditional GET request (cGET) towards the next level to the origin web server. If the object wasn't changed since the timestamp which is typically send with the cGET request (If-modified-Since request), the web server[2] answers with a Not-Modified message and the browser takes the still actual copy of the requested object from the local cache. If the object was changed in the meantime, the origin server directly answers the request with the modified object. After receiving the new object, the browser stores a copy of it in the local browser cache and shows it to the user. For the new cached object the caching policies will apply again. If the policy is client sided then the browser may change the time-to-live due to observations. If the origin web server doesn't want the object to be cached because it's frequently changing, the time-to-live can be set to zero. That means that the object is uncacheable.

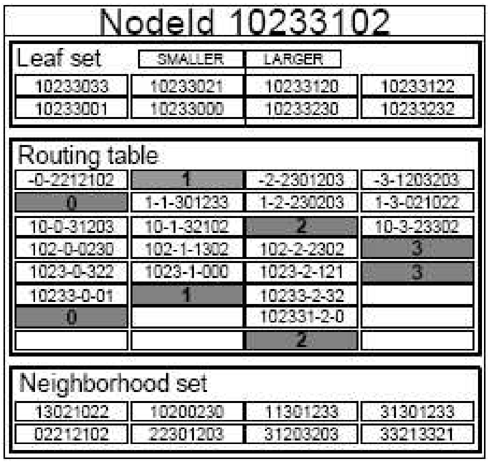
## 2.2 Pastry

Every Pastry node is obviously identified by a 128 bit NodeId. The NodeIds are taken from a circular NodeId space. The set of the NodeIds should evenly be distributed in this space. The distribution of the NodeIds is task of the application, which builds up on Pastry. E.g. it could be the cryptographic hash value of the public key of the node or its IP-address. The objects get ObjectIds the same way, e.g. by hashing the URL or the content to get the 128 bit identifier. Because Pastry exploits network localities to reduce the network load, the distance between two nodes in the network must be

[1]The freshness of an object depends on the caching policy used by Squirrel. Typically the same caching policy as the one used by web browsers will be taken. This can be a time-to-live (TTL) e.g. provided by HTTP fi elds (MAX_AGE, EXPIRES) or a heuristic or whatever

[2]Or the next web cache towards the origin server, respectively

measurable. This can e.g. be realised by counting the number of hops between the nodes.



NodeId 10233102

Leaf set

| SMALLER | | LARGER | |
|---|---|---|---|
| 10233033 | 10233021 | 10233120 | 10233122 |
| 10233001 | 10233000 | 10233230 | 10233232 |

Routing table

| | | | |
|---|---|---|---|
| -0-2212102 | 1 | -2-2301203 | -3-1203203 |
| 0 | 1-1-301233 | 1-2-230203 | 1-3-021022 |
| 10-0-31203 | 10-1-32102 | 2 | 10-3-23302 |
| 102-0-0230 | 102-1-1302 | 102-2-2302 | 3 |
| 1023-0-322 | 1023-1-000 | 1023-2-121 | 3 |
| 10233-0-01 | 1 | 10233-2-32 | |
| 0 | | 102331-2-0 | |
| | | 2 | |

Neighborhood set

| | | | |
|---|---|---|---|
| 13021022 | 10200230 | 11301233 | 31301233 |
| 02212102 | 22301203 | 31203203 | 33213321 |

Figure 1: Pastry node state

Each Pastry node maintains 3 state sets, a leaf set, a neighbourhood set and a routing table (Figure 1). The leaf set contains L nodes, the nodes which are numerically closest to the Pastry node, with respect to the NodeId. Half of these L nodes have a NodeId smaller than and half of them have NodeIds bigger than this Pastry node. The neighbourhood set contains the nearest nodes with respect to the network distance. The routing table is realised with prefix routing and contains for each place in the NodeId space one line. Line *n* e.g. contains a list of nodes which have the first n parts in common with the prefix of the current node, but not the *n+1* part. If there are more nodes which satisfy this condition then the nearest one is written into the routing table.

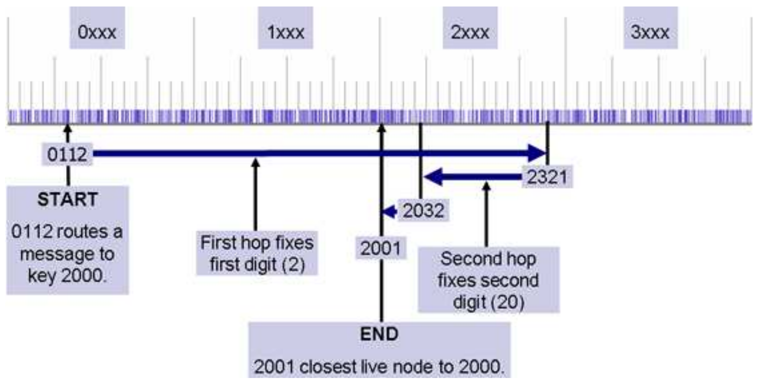

0xxx
1xxx
2xxx
3xxx
0112
2321
2032
2001
START
0112 routes a message to key 2000.
First hop fixes first digit (2)
Second hop fixes second digit (20)
END
2001 closest live node to 2000.

Figure 2: Pastry routing example

If there is a message that should be routed to a given NodeId, the node checks if the ObjectId of the message is in its leaf set. If this is the case the massage will be forwarded directly to the target node. Otherwise the message will be forwarded to the node with NodeId closest to the ObjectId, step by step. This is done with the principle of prefix routing, so the message will always forwarded to a node which has a prefix at least one part longer in common with the message (Figure 2), with respect to the current node. If no node exists with this attribute the message will be forwarded to a node with the same common prefix length than the current node and which's NodeId is numerically closer to the ObjectId.

Given a network consisting of N nodes, then Pastry can route a message within $\log_{2^b} N$ steps[3] to the numerically closest node in the network, where b is a configuration parameter with the typical value of 4.

As long as there are not L/2 neighbouring nodes that fail simultaneously it is guaranteed that the message will arrive at the destination. L was the size of the leaf set. Because this algorithm minimizes the distance in the network and lookups can be executed very efficient, this system scales very good, compared to unstructured networks.

[3]This is valid for the "normal case"

# Chapter 3

# Squirrel

Squirrel uses Pastry[9] as a location and routing protocol. When a client[1] requests an object it first sends a request to the Squirrel proxy running on the client's machine. If the object is uncacheable[2] then the proxy forwards the request directly to the origin Web server. Otherwise it checks the local cache. If a fresh copy of the object is not found in this cache, then Squirrel tries to locate one on some other node. To do so, it uses the distributed hash-table and the routing functionalities provided by Pastry. First, the URL of the object is hashed to give a 128-bit ObjectId from a circular list. Then the routing procedure of Pastry forwards the request to the node with the NodeId (assigned randomly by Pastry to a participating node) numerically closest to the ObjectId. This node then becomes the home node for this object. Squirrel then proposes two schemes from this point on: home-store and directory schemes.

## 3.1 Home-store model

In the home-store model the requested objects are also stored at the home nodes. If a node in the network requests an object and doesn't have a fresh copy in the local cache, it sends a request. This request is send to the home node of the object. The home node can be found by simply mapping the object to its ObjectId using the hash function and forwarding the request towards the closest live node to this ObjectId. The node that can't forward the request to another node which's NodeId is closer to the ObjectId than its own NodeId will finally notice that it is the home node for this object. The home node checks if it has a fresh copy of the requested object already in its local cache.

Given the case it has a fresh copy in its cache. The home node will then send the object directly to the node which issued the request or it sends the client a not-modified message depending on which action is appropriate. This node saves the retrieved object in its cache and returns it to the user.

The other case where the home node doesn't have a copy of the requested object has a stale copy in its local cache is different. The home node issues a GET or cGET request, respectively, to the origin web server hosting the internet object. Then the home node either receives a cacheable fresh copy of the object or a not-modified message. Afterwards the home-node takes the appropriate action with respect to the client,

[1]The terms "node" and "client" will be used interchangeably

[2]An object can be considered uncacheable if, for example, its URL contains "cgi-bin", or if its freshness lifetime is zero (see e.g. [3] for details)

sending a copy of the object or a not-modified message to the requesting client.

Figure 3 shows a simplified version of the home-store scheme.

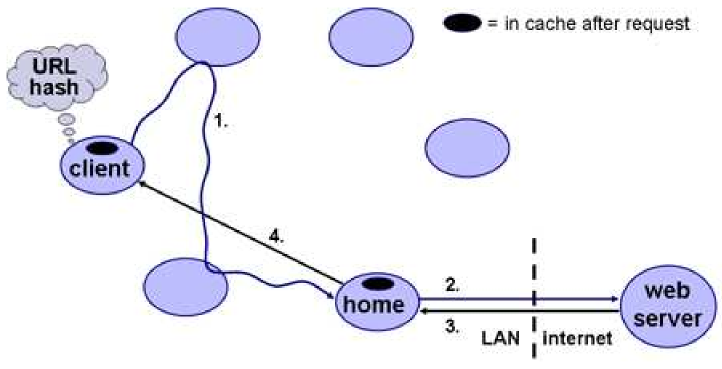


Figure 3: Home-store model

## 3.2 Directory model

In the directory scheme the home-node for an object maintains a small directory of pointers to nodes that have recently accessed the object. Subsequent requests for this object are redirected to a randomly chosen node of this directory (called the delegate), expecting that they have a locally cached copy of the object.

The home node does not store the objects for which it's responsible for locally, it only keeps metadata like the fetch time, last modified time or explicit time-to-live. With this metadata the home node is able to apply the expiration policies of the web browsers without storing the object itself. Therefore, the home node is able to invalidate all nodes with a cached copy from the directory, if the object has changed, at once.

Since the home node does not send requests to the origin web server, it receives the metadata for the object from the delegates which requested the object. A delegate that receives a forwarded request from the home node has to check the freshness of its cached copy before it sends the object to the requesting client. If the object has changed then the delegate informs the home node to update the directory of the object.

Figure 4 is an example for the directory scheme.

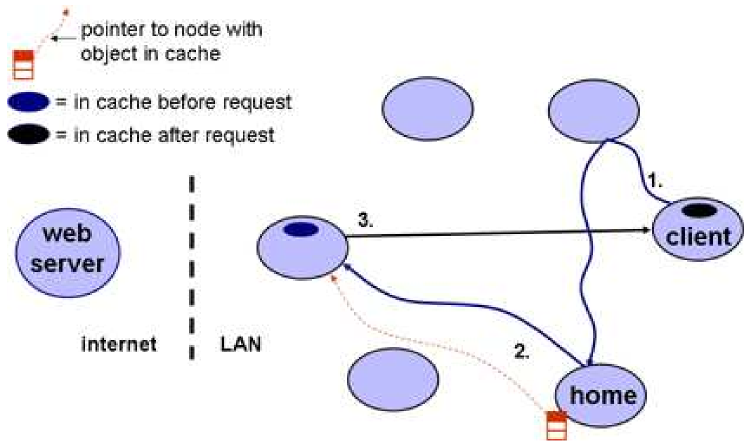

pointer to node with object in cache
= in cache before request
= in cache after request
web server
internet
LAN
3.
1.
client
2.
home

Figure 4: Directory model

## 3.3 Node failure, departure and arrival

Like in any peer-to-peer system, in a Squirrel network, clients arrive and depart the system at random times. There are two different failures of nodes, abrupt and announced failures. Each failure has a different impact on Squirrel's performance. An abrupt failure will result in a loss of objects. To see this, assume that node *N* is the home-node for object *O* and it has a fresh copy of *O* in its cache. If node *N* fails, Pastry will route the next request for object *O* to a new home node *N**. This request will result in a miss if *N** has no fresh copy of *O*, which is likely the case. So the failure of node N will decrease performance since node *N** will have to contact the origin server to get a new copy of object *O*. If a node is able to announce its departure and to transfer its content to its immediate neighbors[9] before leaving Squirrel (announced failure), then no cached content is lost when the node leaves.

When a node joins Squirrel then it automatically becomes the home node for some objects but does not store those objects yet. In case a request for one of those objects is issued, then its two neighbors in the NodeId space transfer a copy of the object[3], if any. Therefore, we can consider that there is no decrease of performance in Squirrel due to a node arrival, since the transfer time between two Squirrel nodes is supposed to be at least one order of magnitude smaller than the transfer time between any given node and the origin server.

## 3.4 Comparing home-store and directory model

The main differences between the two proposed schemes are shown in the Table 1.

| | Home-store model | Directory model |
|---|---|---|
| Store object in local cache | yes | yes |
| Store object at home node | yes | no |
| Home node fetches file | yes | no |
| Client fetches file | no | yes |

Table 1: Differences between the two schemes

The two approaches represent two extremes. There are also attempts to combine the two approaches and take the advantages from both to eliminate the disadvantages. The resulting hybrid schemes achieve better results in term of hit ratio and node load (see e.g. [10]). In return to the attained improvement the implementation of these hybrids are more complicated to realise.

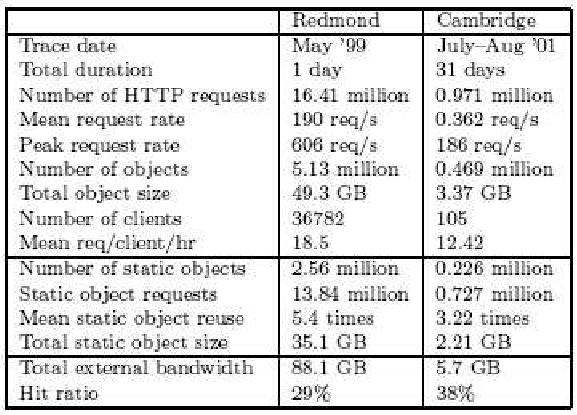
[3]This is easy because each Pastry node keeps track of its immediate neighbors

# Chapter 4

# Evaluation

## 4.1 Trace characteristics

The two traces chosen by the authors of the Squirrel paper are quite different from each other. One with 105 nodes (Microsoft Cambridge) and one with 36782 nodes (Microsoft Research Redmond Campus). The scale of these traces is what the authors wanted to show for Squirrel, that it's applicable for corporate networks with between 100 and 100.000 nodes. Both traces have in common that approximately the half of the requested objects were cacheable. This is a nice fraction, since more and more personalized and dynamic web pages are not cacheable (more to this in section 5.1). To get comparable results with this two different traces, the small one (Cambridge) was collected over 31 days whereas Redmond was collected over 1 single day (because it has quite more nodes).



| | 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% |

## 4.2 External bandwidth

The external bandwidth is defined as the number of transferred bytes between the corporate LAN and the internet. The evaluation in the paper shows that the home-store approach makes more effective use of the available cache storage than the directory approach, despite the fact that the home-store model duplicates the objects requested by the clients in the home node cache.

Both schemes have in common that they perform better the more amount of disk space each Squirrel node provides for the decentralized cache. With only little space per node the directory scheme is slightly better than the case where no web cache is used at all while the home-store scheme performs already obviously better. That's reasonable because in the directory scheme only active nodes which request internet objects cache them. If the available cache size is small, old objects will be deleted to store new objects. When another client now requests an already deleted object, the pointer in the object's directory will be invalid and the object must be fetched from the origin server again. This doesn't happen in the directory scheme that quickly because the objects are not only distributed among the active nodes like in the directory model, but they are spread equally among all nodes (due to the hash function).

With e.g. 100 MB cache size per client both schemes perform comparable to a centralized web cache with enough storage.

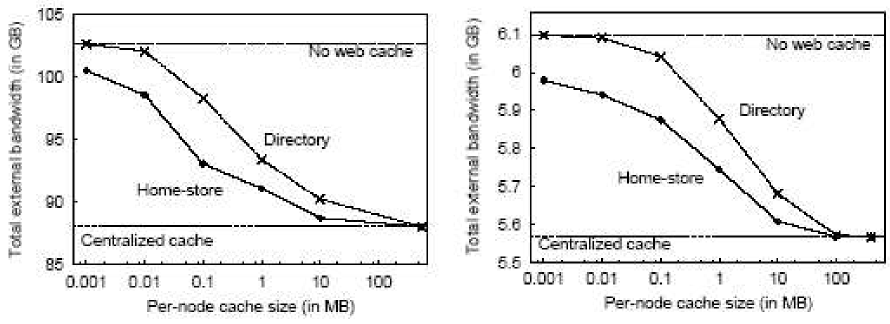


Figure 5: External bandwith for Redmond (left) and Cambridge (right)

## 4.3 Latency

Because Squirrel acts within a corporate LAN located in a single geographic region, the communication latencies in it are a few milliseconds long and at least an order of magnitude smaller than the external latency. The processing time for a request is considered to be only a few milliseconds, too. For large objects the higher inner LAN bandwidth reduces the overall latency.

A request to a centralized web cache always takes two LAN hops, one to the cache and one back again. Squirrel needs a few hops more. This depends on how many hops Pastry needs to find the home node. For the home-store model we need $\log_{2^b} N$ hops to the home node and one back. The directory model needs one additional hop to forward the request to a delegate, in case that there is already a directory for the requested object. The resulting average hop values are presented in Table 2.

| | Home-store model | Directory model |
|---|---|---|
| Redmond | 4,11 | 4,56 |
| Cambridge | 1,8 | 2,0 |

Table 2: Average number of LAN hops

The latency induces by Squirrel is small (at most 15ms) when the object is found in the cache and not noticeable when the object must be fetched from the origin web server.

## 4.4 Load on single nodes

The quintessence of the analysis of the load on each Squirrel node over the whole trace is that the home-store approach is superior to the directory approach. Because the home-store model balances load better there is never a single node that has to serve more than 10 requests per second. This holds for both traces, which speaks for the scalability of the system. With the directory scheme, single nodes sometimes have to serve more than 50 requests per second (Figure 6).

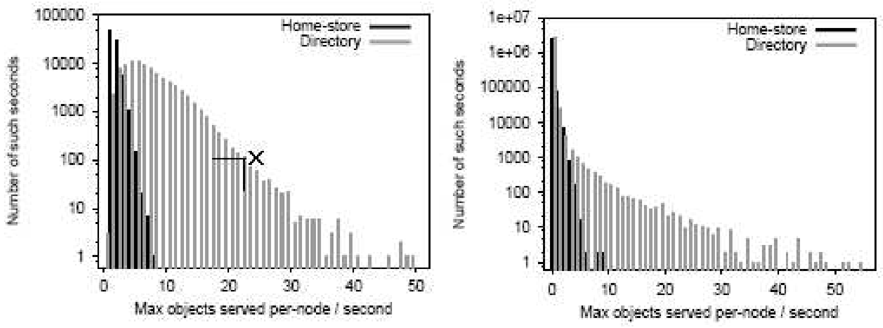


Figure 6: Load distribution for Redmond (left) and Cambridge (right)

When looking at the average load during any second it stands out that the load is extremely low on each node, 0,31 objects per minute and node for both schemes in Redmond. This indicates that Squirrel runs with a negligible fraction of total system resources.

## 4.5 Fault tolerance

In case of link failures that split the network into two sets Squirrel will reorganize into two separate networks whereas the centralized web cache will only be able to serve the requests of the clients in its own subset of the original network. This indicates that the decentralized web cache is more tolerant to failures than the centralized one. In the worst case of a centralized web cache with a single machine, its failure would result in the loss of all cached content. Squirrel nodes, usually desktop machines, are expected to fail or be switched off often. Unannounced node failures result in a partial loss of cached content. This loss is 3,5% of Squirrel's cached content for the home-store model at worst, the directory model on the other hand can lose up to 20% for the Redmond trace. So the directory model is to some extent more vulnerable to node failures than the home-store scheme. After the Farsite study[2], 15% of the machines are switched off every night and/or weekend. This doesn't mean that about this fraction of cached content gets lost, because gracefully shut down machines can transfer popular objects to their neighbors (the new home nodes for the objects).

# Chapter 5

# Related work

## 5.1 Dynamic content

In the last five years a fast growth of dynamic and personalized web content was observed[5, 11]. Under the conventional URL-based caching point of view, like Squirrel's, this trend will reduce the cacheability significantly. So there is need to have another caching concept, not URL-based but content-based. Recent studies[7, 14] have shown that dynamic objects have large portions of data in common. E.g. the study[13] has provided several methods to cache dynamic web contents.

This data provides an opportunity to improve the cacheability by content-based caching algorithms. One example for such an algorithm is Tuxedo[12].

## 5.2 Changed access patterns

The rapid increase in the numbers and types of web servers and clients' diverse interests results in the need for enlarging the cache. For example, Barford et al.[1] give a comprehensive study of the changes in web client access patterns based on the traces collected from the same computing facility with a similar nature of the user population separated by three years. Their experiments show that, compared with the data three years ago, the hit ratios are reduced and the most popular documents are less popular in the transfer data set. This implies that accesses to different types of web servers have become more evenly distributed.

## 5.3 Churn resistant decentralized caching

Churn arises from continued and rapid arrival and failure (or departure) of a large number of participants in a peer-to-peer system. This will increase host loads and block a large fraction of normal insert and lookup operations in the system. The paper from Linga et al.[8] focuses on this problem and studies a cooperative web caching system that is resistant to churn attacks. Based on the Kelips peer-to-peer routing substrate[4], which consists of virtual subgroups and runs a low-cost background communication mechanism, 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 to ensure high cache hit rates even when the system is stressed under churn. The system also improves on the network locality of cache accesses in previous web caching schemes.

# Chapter 6

# Conclusions

It is possible to decentralize web caching in a corporate LAN. Moreover it's cheaper with respect to dedicated hardware and administration because Squirrel is highly scalable and self-organizing. The home-store scheme is the one to be used in reality and approaches the performance of a centralized web cache with infinite storage while using e.g. 100 MB cache size per node. This is achieved with little overhead in the network, on average. There are a few points where future work should bring up new ideas, e.g. to be able to cache dynamic or personalized content.

# Bibliography

[1] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1-2):15–28, 1999.

[2] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43. ACM Press, 2000.

[3] Edith Cohen and Haim Kaplan. The age penalty and its effect on cache performance. pages 73–84.

[4] Indranil Gupta, Kenneth P. Birman, Prakash Linga, Alan J. Demers, and Robbert van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *IPTPS*, pages 160–169, 2003.

[5] Roger A. Horn and Charles R. Johnson. *Matrix analysis*. Cambridge University Press, 1986.

[6] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 213–222. ACM Press, 2002.

[7] Terence Kelly and Jeffrey Mogul. Aliasing on the world wide web: prevalence and performance implications. In *WWW '02: Proceedings of the eleventh international conference on World Wide Web*, pages 281–292. ACM Press, 2002.

[8] Prakash Linga, Indranil Gupta, and Ken Birman. A churn-resistant peer-to-peer web caching system. In *SSRS '03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems*, pages 1–10. ACM Press, 2003.

[9] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.

[10] Bo Sheng and Farokh B. Bastani. Secure and reliable decentralized peer-to-peer web cache. In *IPDPS*, 2004.

[11] W. Shi, E. Collins, and V. Karamcheti. Modeling object characteristics of dynamic web content. *J. Parallel Distrib. Comput.*, 63(10):963–980, 2003.

[12] Weisong Shi, Kandarp Shah, Yonggen Mao, and Vipin Chaudhary. Tuxedo: A peer-to-peer caching system. In *PDPTA*, pages 981–987, 2003.

[13] Huican Zhu and Tao Yang. Class-based cache management for dynamic web content. In *INFOCOM*, pages 1215–1224, 2001.

[14] Zhaoming Zhu, Yonggen Mao, and Weisong Shi. Workload characterization of uncacheable http content. In *ICWE*, pages 391–395, 2004.