The Kraken Distributed WebProxy

Joe Foley, Amy Vandiver, and Ben Vandiver

Abstract— **We propose Kraken[1], a distributed caching scheme which utilizes self-healing methods to reconstruct the distributed cache in the event of failure. Kraken is a managed cooperative cache appropriate for placement near clients which makes a minimal set of assumptions about its environment. No fixed point or statically designated server manages the cache, thus the cache is robust to the failure of any participant. Due to the cooperative nature of Kraken, its performance improves as more users participate. We have successfully implemented such a system as a proof of concept.**

Keywords— **Distributed Web Cache, Web Proxy, Web Cache, Election.**

I. Introduction

The World-Wide Web has steadily grown in popularity since it's inception, and there are no indications that the volume of traffic is decreasing any time in the near future. Network bottlenecks and high server load are both causes are of long delays for users waiting to download pages from distant servers. Caching has proven an effective means of reducing end user latency and network bandwidth consumption[4]. Currently caches are commonly found incorporated into browsers and web-proxies.

Web browsers contain memory and disk caches because clients often request the same information repeatedly. Both because users tend to backtrack and web sites contain duplicate references to images on multiple pages on the site. By placing a cache between a set of clients at the Internet, more sharing can be detected and exploited to reduce end user latency.

A standard solution to this problem is to run a caching web-proxy such as `Squid` at the entry point. However, this creates a single point of failure since most caching-proxy schemes assume that a user only connects to that particular proxy and no other. In the case of that server going down, all clients associated with that server are unable to discover a new proxy. In addition, a squid server is by necessity a dedicated special-purpose machine since it must handle all of the web-requests of that subnet. Any system which has a centralized server has the inherent problem of scalability. As the number of browsers increases the draw on server resources increases, and thus the performance drops. This is even true of a scalable clustered web-cache system. The standard answer to this, is purchase another server to add to the resource pool.

A solution to these problems is to enable individual client machines to share their caches in a cooperative manner. Unlike file systems, where sharing a cache among a number of machines introduces a large amount of complexity due to cache coherence, the properties of web document allow for a much simpler solution. This simplification is possible for two reasons: First, data is read frequently and written infrequently by a limited number of sources, thus concurrent updates are not a problem. Secondly, a worst case failure means the cache delivers stale data, and the end user can always hit “reload” and obtain the most recent page.

In a cooperative caching scheme, the cache size and CPU power increases as more clients join the cooperative cache. However, this introduces a discovery problem: is the page you want in the cache, and if so, where is it? Solving this problem involves solving the coordination problem.

A. Problem Statement

We are trying to solve the problem of coordinating cooperative caching among an unmanaged set of clients over a local area network. We assume that clients are joining and leaving the cache regularly and that they are not centrally managed or coordinated. Additionally, clients are allowed to fail at any point. Finally, no client is privileged above the others with respect to inter-proxy coordination.

The network connecting the clients is assumed to be relatively fast and high-bandwidth compared to the regular traffic; thus leaving space for additional communication between clients. Furthermore, we assume that network partitions do not occur. We also do not assume that broadcast messages are supported by the network or cheap. Since the network is not assumed to be of a broadcast nature, snooping does not allow a client to acquire information about what the other clients request.

We decided to focus on the problem of coordinat-

[1]Named for the mythical giant squid of yore.

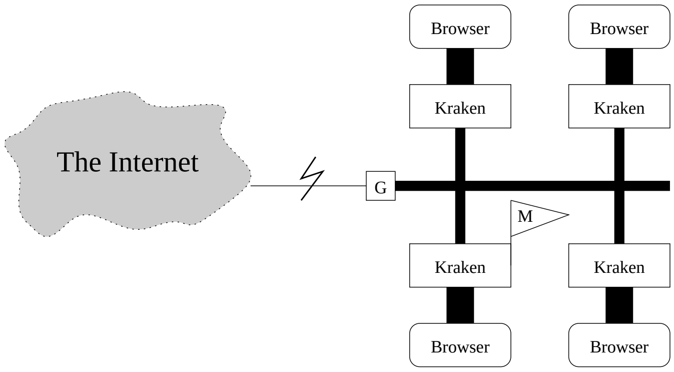


Fig. 1. Conceptual drawing of standard Kraken deployment.

ing the clients in order to merge them into an effective whole. The resulting system should achieve good performance in the absence of failures. In the presence of failures, the system is allowed to degrade in performance until the failure is resolved, at which point it should return to previous levels. There are numerous other pieces related to minimizing network traffic and optimizing the cache replacement policy which have been investigated previously and could be applied with some modification to our solution to improve its performance.

II. Related Work

The idea of combining multiple caches into one cooperative cache is not new. There are several models of ways to enable caches to work together: peer to peer, hierarchies, and segmented namespaces. A distributed caching solution must provide a means for page discovery in the cache, a replication and replacement policy, and a manner in which a cache may be added or removed from the distributed cache.

A. Peer to Peer

Peer to Peer caches implement the simplest method of page discovery: ask everyone. On a local miss, the cache asks all of the caches in the distributed cache concurrently. If it receives a positive response from another cache, it fetches the entry and returns it to the client. In the event that none of the other caches has the page, the cache waits for all the negative responses, then visits the server for a copy. The replacement policy is localized to each individual cache and every cache replicates every page it encounters. The problems with this approach are obvious: it generates a lot of network traffic, an even longer latency on a miss, and it scales very poorly. Thus by itself, peer to peer caching is not a reasonable caching solution.

B. Hierarchical

Hierarchical Caches are organized into trees, using a parent-child relationship. Numerous implementations of hierarchical caches are in use[7], [4], [12], each including a different set of optimizations on the base concept. When they miss in their local cache, they ask their siblings and their parents, returning the first hit, or going through the appropriate parent to fetch the page from the server. This system provides cooperation at the cost of a very structured and inflexible cache structure. Moreover, to ensure good performance, each cache must implement a contract that states what it will be responsible for. Errors in this contract can destroy the performance of other caches in the hierarchy. ICP, the Internet Caching Protocol[8], is used by hierarchical caches to provide a standard for requesting and exchanging cache entries between hosts in a distributed cache. The replacement and replication policy for hierarchical caches is the same as that for peer to peer caches. Finally, new caches are added by manually editing the parent, child, and sibling tables of the current caches.

C. Segmented Namespace

By segmenting the namespace for pages into chunks and distributing it among the caches, the page discovery problem can be reduced to a name lookup. In a segmented namespace scheme, each cache is responsible for a small piece of the total cache. The standard method of mapping pages onto a sharable namespace is by computing hash function of the URL. The clients are directly responsible for computing the hash of the desired page and then translating the result into the address of the cache to contact. This scheme pushes page discovery into the client, freeing the caches from this responsibility. Replacement is handled locally, while replication is determined by the translation method. The problem with this approach is when caches are added or removed from the system. At this point, all clients hashing functions must be updated and entries moved from one cache to another.

A number of implementations attempt to solve this problem through a better hashing algorithm[9]. Karger and Sherman[11] present a hashing scheme that pushes the translation onto the DNS servers. Under consistent hashing, the client computes a hash of the URL that maps it to one of 1000 virtual caches. The client requests the desired page from one of the virtual caches and the DNS servers route it to the appropriate place. This scheme removes the discovery problem at the cost of a extra work to move sections of the URL-space around when new caches are added or removed. Due to the manner the hash is implemented, not many entries in the cache need be moved, which represents a significant improvement over other hashing schemes.

Since Kraken attempts to work with minimal assumptions about user powers, DNS hacking is not a viable solution. On the other hand, the use of a good hash to map to a segment of the URL-space to a manager is something very useful and could be incorporated into the multiple-manager case of Kraken.

D. Akamai

Akamai reduces the cost of downloading a web page from a distant server by attempting to cache the images on the page closer to the clients. Since Akamai only caches images, the latency to retrieve the hypertext document can still be very large. By using a proprietary means to efficiently replicate and route connections to their servers, Akamai attempts to minimize the distance between the browser and data source to decrease latency and increase effective bandwidth. However, if the bandwidth of the connection between the client and the closest Akamai site is low, retrieving a web page will still take a sizable amount of time. Consequently it is important to cache web pages on the client side of a low bandwidth connection.

III. Design

A. The Idea

In order to solve the page discovery problem, our system, Kraken, promotes one of the clients to a managerial position. The manager keeps track of the mapping between URLs and which clients store them. Thus, when a client wishes to discover if a page is in the cache, it asks the manager and the manager supplies the location of the client that caches it. By introducing a manager, the act of discovering the location of something in the cache takes constant time in terms of number of participants as opposed to using a scheme more like FreeNet[2], [3] which takes logarithmic (or worse) time. However, just like FreeNet, the manager is not guaranteed to know whether the page is actually in the cache; it just gives out hints as to where it believes the page resides.

A centralized manager also allows for more global optimization of cache entry placement. Specifically, cache entries can be replicated around the cache based on load so no one proxy serves most of the load on the cache. Cache entries can also be moved closer to the clients that request them the most often, further reducing latency. Finally, if cache entries can be forwarded instead of just flushed, a central manager can better identify cache entries that should be discarded versus cache entries that should be forwarded (singletons)[16].

B. The Issues

Introducing a manager introduces a number of new problems. First, when the cache is constructed, a client must be selected to assume the manager role. Second, the cache must be able to regenerate a new manager in the event that the manager fails. With a fixed or known set of clients, these problems are relatively trivial, but they quickly become more complex in the face of uncertainty. Thirdly, the clients must be able to discover the manager when they join the cache. Performing a broadcast operation whenever a client joins is infeasibly expensive with a large number of possible clients, thus an incremental or probabilistic method of manager discovery is necessary.

Since a single manager is now a point of failure as well as a performance bottleneck, a related issue is that of replicating the manager to avoid these problems. Since the objective of Kraken is to produce a system that scales with the number of clients, introducing a new bottleneck fails to accomplish this objective. However, manager replication produces a whole new set of complicated issues. These issues and a set of proposed solutions are discussed in sectionIII-C.2.

C. The Solution

Kraken solves the intra-client discovery problem by having each client maintain a list of neighbors. When a client starts up, it asks its neighbors to find out the current manager. If it does not receive a response from any of its neighbors, it becomes the manager. Should it hear conflicting responses from its neighbors, it joins the cache of one of the managers it heard about and tells the managers to hold an election. The election algorithms are given later. Presuming a reasonable set of neighbors, this provides a reasonable chance of producing a completely connected set of clients.

When installing Kraken, the only setup necessary is to tell your client about one system currently running Kraken. Your client can then ask this Kraken who the manager is. The manager will then provide your Kraken with a complete neighbor list. This setup process is vastly simpler than that of current solutions such as Squid.

When a client notices a manager failure, it first asks its neighbors to determine if a new manager has been selected. If it hears nothing back except the original manager, it declares itself the manager and informs its neighbors of its new status. Upon receiving such a message, the clients first check to see if the original manager is still up to prevent malicious client-stealing. If the original manager is still up, the client ignores the message. Should one of the neighbors have concurrently realized the lack of manager and declared itself the new manager, an election is held to resolve the issue. This method accomplishes a grass-roots reconstruction of the cache that utilizes the same method as the construction of the original cache, thereby reusing code and reducing complexity.

The manager reconstruction process is slow and involves sending a lot of data over the network, which reduces performance until the reconstruction completes. The actual cost in terms of number of election messages for this reconstruction is order the number of clients times the square of the number of neighbors per client. Since the re-elections are done pairwise, the cost of an election is constant in this case. We assume that managers fail infrequently, thus allowing for such a costly rebuild operation. When a manager is shutdown gracefully, it designates one of its clients to be the new manager, transfers the managerial data to that client, and informs all of the clients of the new manager.

One major issue that has not been previously addressed is the problem of security in a cooperative cache. Most people consider their web requests to be fully known by people attempting to snoop them. Current sniffing programs fully reveal the requests other people on the local network make. Thus, the issue of other people knowing what has been hit recently is less critical. The Kraken system does allow this information to be more easily accessed, but it also spreads out the location of the information. Since the manager is not designated or necessarily static, the access log is not available unless all managers collaborate to reproduce it. Caches are already forbidden from caching encrypted or authentication-protected pages, thus privacy of private documents is not an issue. A potentially more serious problem is when your cache stores a page which someone else downloaded. This would occur if a page-forwarding system is used to ensure that page remain in the cache longer. The legal issues of whether people are responsible for data their machines store is fuzzy at best and changes depending on your location.

C.1 Elections

The most interesting part of this project lies in its election scheme. Elections occur when a proxy starts up and receives conflicting responses when it asks its neighbors who their manager is. In the single manager case, the winning manager continues running without interruption. The losers transfer their directories (URL to client mappings) and client lists to the winner. After the transfer is complete, the loser sends a message to all its clients informing them of their new manager. Should a request come to it after this point, it will redirect the request to the real manager.

An election algorithm results in all the participants knowing that a specific participant in the winner. The topic of election algorithms have been well researched[6]. A prerequisite of most election algorithms is that each participant must have a unique ID. Normally the winner is selected by maximizing some fitness criteria with ties being broken by the unique identifier. Election algorithms also assume some knowledge of the arrangement of the network, either the exact topology such as a ring or the maximum number of network hops to reach any other host (network diameter).

In the case of Kraken, the network topology is assumed to be a local area network with no partitions, thus it is a fully connected graph. The unique identifier was chosen to be the IP address of the machine and the port number of Kraken. Kraken uses a slightly modified FloodMax scheme[6] to select a new manager, with a bias towards the manager with the largest number of clients. The total number of messages sent as part of the election is proportional to the square of the number of participants. However, the maximal number of participants in an election is the maximal number of neighbors of any one client. If the number of neighbors per client is the logarithm of the total number of clients, the cost of an election is the square of the log of the number of clients.

For example, Proxy A starts up and asks its neighbors who the manager is and discovers both manager 1 and manager 2. Proxy A sends a message to both manager 1 and manager 2 telling them to hold an election. Then managers send each other their number of proxies. They both calculate who has more clients, or if there is a tie, who has the lower IP address and port number. The loser sends the winner its directory and client list.

If Proxy B starts up and discovers managers 2 and 3, and sends the election message to 2 and 3 before the election between 1 and 2 resolves, there is a potential problem. One solution would be for manager 1 to forfeit the second election entirely, thus resulting in manager 2 and it's proxies remaining separate from distributed cache of manager 1 and 3. We do not think this in an acceptable solution because in the event of a large failure, it is probable that original distributed cache will not reform and instead multiple smaller distributed caches will result. The next most simple solution would be to have manager 2 wait for the first election to resolve before responding to the second election. However, if a cycle of blocked elections occurs, deadlock will result.

To resolve the multiple elections issue, a method of breaking cycles is needed. Our election algorithm breaks cycles by preempting "lesser" elections. The key idea is to use some function that a client can compute given only the election participants, which orders all possible sets of elections. Since the ordering covers all possible sets of elections, a dependency cycle will always be broken. If election {A,B} cannot preempt {B,C}, and {B,C} cannot preempt {A,C}, then election {A,C} preempts both {A,B} and {B,C}. Thus, {A,C} will resolve regardless of the ordering of the initiation of the elections.

When an election is preempted, the client must remember the members of the previous election. When the election resolves, the sets of people involved in preempted elections are sent to the winner. The winner then re-initiates an election with the preempted parties. This solution does involve participants storing non-replicated information, thus if a participant, the cache may not resolve to a single entity. However, this situation involves a client dying at exactly the right time in order to produce a situation that could occur anyways. Therefore, the election algorithm results in a single cache with a single manager, barring bizarre failure modes.

C.2 Multiple Managers

The basic design decision for multiple managers is deciding which managers store what data. If all managers are equally able to satisfy requests, the URL to client mapping must be replicated over the whole set of managers, introducing a lot of manager crosstalk and coherence problems. Even though strict coherence is not necessary, the overhead is proportional to the number of client misses times the number of managers. The other option is to segment the URL space and assign a portion to each manager. This method requires less communication between managers, but requires clients to maintain the mapping of segments of the URL space to specific managers. Additionally, in the event of manager failure, a section of the URL to client mapping must be reconstructed. Consequently a combined approach involving a small amount of replication can reduce the cost of manager reconstruction after failure without dramatically increasing the overhead during normal operation.

There are several possible methods to map URLs to managers. The first method is to use a consistent hash as recommended by Karger and Sherman[11]. Alternatively the "table" that maps segments of the URL space to mangers can be stored in a tree with the manager data located on the leaves[10]. In order to look up a specific URL, the initial bytes of the hash are used to walk the tree. This tree structure allows managers to be responsible for different size segments of the URL space thus providing an easy mechanism for load based partitioning. If a manager is overloaded, it simply divides the URL space it is responsible for in half and promotes one of the proxies to be the new manager of half of it's URL space. Replication among managers is also easily supported by this structure.

One method of achieving fault tolerance in the face of manager failures is to arrange the managers in a directed ring. Each manager is responsible for manager below it, and should be able to detect that the manager below it failed within a reasonable length of time. If the manager below it fails, the manager above it either takes over the URL space itself or promotes a proxy to be the new manager. The new manager must be informed of who is above it, below it, and what segment of the URL space it is responsible for. The new manager mapping is then passed around the ring in the form of a delta. The manager set lazily informs the proxies that a new manager has been added to the mapping between URLs and managers.

Holding elections among multiple rings of managers is another source of great complexity. Each ring elects an election handler. In the case of 2 rings, election handlers would split their rings below them and trade, thus connecting the 2 rings. In the case of many rings, the election handlers use the single manager election algorithm to designate one manager responsible for resolving it into one ring. Proving that it is possible to make this fault tolerant is possible but beyond the scope of this paper.

IV. Architecture

The Kraken system is composed of three major pieces, which all operate together in a tightly coupled manner. First, the *Proxy* acts as the intermediary between the client's browser, the cache, and the Internet. Second, the *Cache* stores the local part of the distributed cache as well as handling client-side discovery and cache entry routing. Lastly, the *Manager* maintains the list of clients and a directory of the mapping between URLs and clients. In a Kraken cache, each client runs all three parts, although the manager is only active in some of the clients.

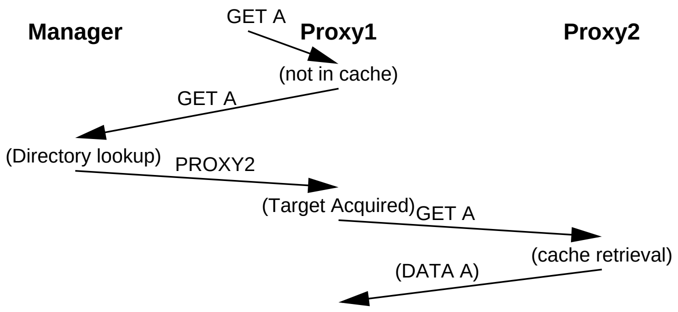


Fig. 2. Protocol Analysis for Simple Kraken Cache Access.

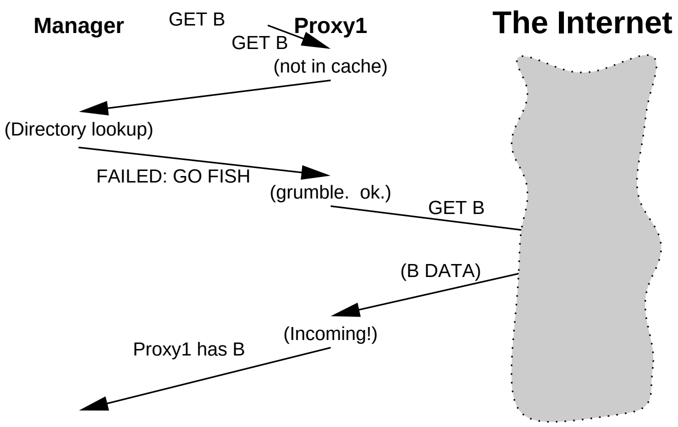


Fig. 3. Protocol Analysis of Kraken Cache Miss.

A. Proxy

A proxy system was chosen as the entry point for clients because modifying the browser itself to include the entirety of Kraken is not feasible. Even if it were, then only users of modified browsers would be able to participate in a Kraken distributed cache. Thus, the use of a proxy produces browser independence and simplicity of design at the cost of some performance. Since most operating systems are relatively intelligent in the manner that they send network traffic to themselves, this additional latency should not be visible to the client.

B. Cache

The cache system of Kraken presents a standard interface to the proxy. It supports the familiar operations of `get`, `put`, and `flush`. In addition to the local proxy interface, the cache also listens on a network port in order to satisfy all inter-cache and manager-initiated operations. The cache serves remote client requests for pages located in the local cache. The cache also responds to `flush` messages from the manager, and notifies the manager in the event that an entry was flushed from the local cache. This design makes it easy to test out different replacement policies without having to make major modifications to the code.

When the proxy receives a request for a cachable page from the browser, it attempts a `get` operation on the cache. The cache first checks the local cache for the object. If it doesn't find it, it issues a request to the manager. The manager either responds with a `failed` message if the object is not in the distributed cache, or with the IP address and port of a client who the manager believes has a copy of the page. If the lookup failed, the cache returns a miss to the proxy and the proxy visits the server for a copy. If the copy is cachable, the proxy performs a `put` operation, inserts the page into its local cache and informs the manager. If the lookup succeeded, the cache asks the specified client for the page. The client responds with the data if it is it's local cache, or responds `failed`. Failure at this point is treated the same as a failed manager lookup, with a trip to the server to fetch the page.

C. Manager

The manager is responsible for maintaining the client list and the mapping between URLs and the clients which have them cached. A client joins the cache by locating the manager and sending it a `join` message. At this point, the manager has the chance to update the client's neighbor list. Due to the critical nature of these neighbor lists, it is very important that they be accurate and well distributed to ensure the minimal number of elections to construct the complete cache. It is probably necessary for the number of neighbors to be proportional to the log of the number of clients in the cache. It is beyond the scope of the current project to determine the optimal method of assigning neighbors. Neighbors are generally fixed or unmodifiable by algorithms that use them, thus this appear to be a field which has not been researched fully.

The manager currently fields a `get` request for every page that missed in the local cache. This allows the manager to accurately track page usage and direct clients to replicate frequently requested pages to reduce the load on any one client. The current system replicates pages based on the logarithm of the total number of accesses. In addition, the manager uses a round-robin scheme to distribute load equally among the replicas. Finally, when a client caches a newer version of a page than is currently in the distributed cache, the manager sends `flush` messages to all clients who are caching stale copies.

Should a client fail, another client will discover the failure when it attempts to get a cached page. It will then inform the manager that the client has failed. The manager verifies that the client is indeed down, and removes it from the client list. Cache entries stored on failed clients are lazily deleted from the directory when they are accessed, thus avoiding the costly operation of deleting by searching through the entire directory.

V. Implementation

"The Devil's in the details, and it's all details"

For ease of implementation and rapid development, Kraken was written in Java 1.3. We found the Object-Oriented language had many useful features including: network sockets, easily interchangeable streams, and fairly robust error handling. For example, the implementation of the HTTP proxy server as a multi-threaded Java server took under 3 hours.

Java also supplies a well-integrated threading system, which allows for asynchronous servers at the cost of having to write thread-safe code. By minimizing the places where concurrency is an issue, the code was much easier to write. Specifically, a standard producer / consumer object was used to exchange data between the server, its threads, the election system, and the network. However, due to the nature of the problem of many interacting cache entities, numerous concurrency issues had to be resolved. The common failure modes for the multi-threaded code are either to I/O block or to have a client fetch a page from the server unnecessarily. Unfortunately, our testing code runs a large number of Kraken all in the same Virtual Machine, which leads to an increased possibility of thread problems. Before being suitable for wide scale deployment, the threading issue would have to be revisited and all the code examined for blocking and locking conditions and races.

Our specific implementation was tailored for simplicity and speed of implementation, not for performance. This allowed us to investigate the desired features in the time alloted. We made numerous simplifications to make the task easier. The local cache on each machine is implemented as a memory cache of a fixed size with an LRU replacement policy. The cache indexes on URL instead of a hash of the URL. This causes the directory index to be of variable and sometimes sizeable length. Additionally, we chose to implement the protocols over TCP when most of them require only one-way light-weight communication. Finally, we ignored standards like ICP in our implementation for simplicity. Despite these many performance degrading shortcuts, the end product is still usable and demonstrates the key features.

A. Protocols

Each of the modules has a separate protocol for handling its transactions. For simplicity's sake, the protocol has many similarities to HTTP's request-and-reply exchange.

1. client opens connection to server
2. client sends request "<REQUEST> <ARGUMENT> <PROTOCOL>"
3. server sends back data or a formatted response
4. server and client close connection

In fact, the protocols are mostly a superset of HTTP. For consistency, it would be nice to formulate them all in HTTP, which would enable code reuse[8] However, this was not possible and consequently the <PROTOCOL> argument in the request was removed because it is unnecessary in the Kraken.

Proxy

GET	Retrieves data associated with a URL.

InterProxy

GET	Spool entry to another client.
FLUSH	Flush entry from local cache.
WHERE MANAGER	Returns who this Kraken believes is the Manager.
NEWMANAGER	Change managers

Manager

GET	Tells manager to ask where data is cached.
PUT	Notifies manager that cache has a given page.
FLUSH	Removes specified cache entry from directory.
JOIN	Requests addition to directory.
LEAVE	Proxy leaving collective, delete all associated entries.
DOWN	Suspected proxy failure.
KEYS	Returns keys of directory.
CLIENTS	Returns lists of clients in directory.
ELECTION	Tells manager to hold election.
DIRECTORY	Instructs manager to add these directory entries to its directory.

A.1 Election Protocol

The election protocol implements the fault tolerant, unknown participant election algorithm described above. It uses three types of messages: `NEW ELECTION`, `STATS`, and `PENDING`. Each message includes a to, from, and members of the current election. A `STATS` message also contains the number of clients and the list of known managers. The election algorithm is implemented as a state machine which processes and sends election messages synchronously. Since election messages are not common when compared to get requests, no multi-threading is necessary to provide short latency response.

B. Implementation Optimizations

B.1 Hints

One of the major performance problems with the current design is that clients must talk to the manager anytime they want to locate anything. By taking advantage of the structure of the web, a system of hints can be developed to reduce the number of times the client must ask the manager. Specifically, the client examines any HTML document that it caches, looking for URLs. When it later sees a reference to any of those URLs, it remembers the answer it got from the manager about that URL and adds it to a table of hints. The entry in the hint table is composed of the root page, the page it links to, and the location of that page in the distributed cache. Whenever it later serves an inter-proxy request for the root page, it also sends along the the hint table entries.

Before going to the manager, a client checks its hint table to see if it can guess the right location of the page. This guess is based on information provided to it by other clients who knew where it was a while ago. A simpler scheme would be to assume that all the links on a page reside in the same cache that served the page. This eliminates a number of mappings that would be necessary to provide hints. Hints are set to expire after a certain amount of time to prevent truly stale information. Should a hint prove incorrect, the manager would then be queried. If the hint hit ratio exceeds 50%, the hint system wins in terms of network usage. Any hint hit ratio removes load from the manager. Given that HTML pages tend to contain a lot of images, this hint system can significantly reduce the manager load and latency.

B.2 Forwarding

Another feature of cooperative caching is adjusting the cache replacement policy to include the possibility of forwarding an entry instead of flushing it. Due to the managed nature of the cooperative cache, the manager can detect when a client is caching the last remaining copy of a popular page. In this case, the client would forward the entry to another client instead of flushing it; thus keeping it in the cache. When implementing such a scheme, care must be taken to avoid shuffling entries infinitely around the network. Cache entry forwarding has been implemented by a number of distributed file systems[16], and these forwarding schemes could easily be adapted to web caching.

VI. Validation

The purpose of validation is two fold: to verify that the system was implemented correctly, and that the design has the expected performance characteristics. Proving correctness of our system in the absence of failures was relatively easy due to the simple design. However, proving correctness under failure is much more difficult due to the infinite number of ways a distributed system can fail. In particular, the election algorithm which lies at the heart of Kraken's fault tolerance was exercised but not truly verified.

A. Correctness and Fault Tolerance

Our implementation correctly caches documents and takes advantage of sharing to expand the effective size of the cache. Simple test runs and usage were enough to determine that Kraken caches as per the design. In order to more easily test the caching system, the cache was set to ignore the Expires heading, effectively increasing the number of cachable pages.

The replication system was also tested empirically, and it performed well, as each replica is dynamically assigned and acquires its copy from the cache. Thus, each replica need not pay the miss penalty for a cold start.

In order to test that the election algorithm we proposed actually worked, a single-host testing system was created (in Java). This system follows a script which tells it to create a number of Kraken servers on the same machine using incremental ports. Secondary scripts tell created Test clients to query these Krakens for specific documents. By examining the logs, it was possible to verify that specific significant events occurred, and how the system was able to respond.

There are 4 basic test cases that can be used to test for correctness:

1. If a series of Krakens are started one at a time such that each knows about at least one of it's predecessors, the resultant distributed cache should resolve to all clients agreeing that the first Kraken is the manager.
2. If 2 managers start independently, and then a third Kraken starts that knows about both of the managers, the resulting election should be won by the manager with lower identifier (IP address and Kraken port number).
3. If manager A has n clients and manager B has m clients, and a Kraken starts that knows about at least one client of each manager, it should result in an election won by the manager with the grater number of clients.
4. Finally, if the manager crashes, the proxies should elect a one new manager.

Our implementation of Kraken passes all of these tests, in addition to many more complicated cases involving simultaneous start up. If 2 Krakens that know only about one another are started at the exact same time, they are not guaranteed to discover one another. Simultaneous start up can also result in many simultaneous conflicting election messages. However, because elections resolve so quickly we had trouble testing this and actually had to insert delays into the code to be able to verify the correctness of the algorithm.

During a test of a 5 Kraken distributed cache, individual proxies and the manager were repeatedly shutdown with no major problems. The cache always managed to elect one manager, and the system did not hang while trying to serve a page from a failed cache.

B. Performance

In order to demonstrate that our implementation achieved our design performance goals, the implementation had to be instrumented and subjected to normal and abnormal loads. Simulating a heavy workload is very difficult for two reasons: the mechanics of producing a heavy workload, and the difficulty of accurately simulating users. The mechanics are difficult because a good cache implementation can easily satisfy one host , thus truly pushing the envelop requires a bank of machines, all smashing the proxy. However, this only tests the scalability of the system, not its performance under a real workload. Simulating users is difficult because user profiles change over time and users are not very predictable in terms of which pages they hit when. In theory, recordings of user activity could be replayed to reproduce a workload for testing purposes. However, the results of requests are time dependent, even for non-dynamically generated pages. Thus, a recording is only useful for a limited time after it was created. To test a distributed cache also requires the network to be a part of the simulation, as it has a big impact on the performance of the cache.

B.1 Qualitative

The Kraken internal testing system was designed for doing validation testing. It was not really designed for gathering performance characteristics. The added complexity required to make it of sufficient capabilities for high-bandwidth testing would have made it much more complex, and therefore harder to debug[2].

After our initial implementation phase the Kraken system (development time was approximately 1 month), we were able to get it functional enough for informal testing. Even without a number of the optimization schemes we mentioned for improving the caching system, there is a very visible difference in page loading speed. In fact in our perception, there was no distinguishable difference between running a Kraken cache locally on 4 machines, and having 4 browsers point at a common Squid server. Alas, because of personnel limitations and computing resources, we were not able to do wide-scale field testing with many clients to see if there was obvious degradation.

[2]As it was we spent a good deal of time tracking difficult bugs back to the Tester.

B.2 Simple Single-Connection Tests

To make use of real-world data and gather quantitative data, we took a dump of a in-use Squid's logs, extracted the URLs from them and applied a number of transforms upon them (mixing URLs and duplicating) to emulate a generic user. The base parameters for the amount of duplication were garnered from [4], where they said that approximately 70% of all content that is cachable is shared content that could be cached. Quite often only approximately 30% is actually cached. The amount of mixing was determined from looking at a number of user logs and estimating.

All of the tests were done on a Sun Sparc Ultra5 running Solaris 2.7. The workstation has 128MB of memory, a single Sun UltraSPARC-IIi 333 MHz CPU and connected by 10MB/s ethernet to MIT's network. For a simple measure of how well the cooperative caching worked, We compared a single Squid server configured[3] to 10MB of memory cache and 10 MB of disk cache to a pair of Krakens with 10MB of memory cache each. Each was presented with 2 clients (`wget` with the appropriate options in the case of Squid, and an internal test agent for Kraken.)

A qualitative observation that quickly becomes clear when watching the tests run is that the Kraken spends most of it's time on broken links and not on fetching documents. Another observation is that squid has the same problem. In both cases, eventually internal timeouts for the initial TCP connection appear to have been what severed the connection and allowed the testing to continue. This explains the extremely high latency averages and maximums compared to what most results were.

However, these performance characteristics are not really a good characterization of how well the proxies work. The obvious problems are twofold:

1. Most modern web browsers use multiple concurrent connections to load the information at the same time. Wget and our testing instances only open a single connection at a time.
2. When either cache tried to follow a URL for a server that no longer existed, that thread hung until a TCP gateway timeout was reached, causing the maximum and average latency to be huge.

B.3 Multiple Concurrent Connections

Building a testing system capable of both stress-testing a proxy and/or emulating standard patterns of use is beyond our short-term capabilities. Thankfully, many other researchers have been investigating this problem too, and have developed a variety of tools. Each has its own feature set and limitations.

- SURGE[4][14] Creates a number of calibrated files, then fetches them using a client to generate statistics.
- Web Polygraph[13] Uses an artificial client and server to do traffic flow
- CacheFlow performance testing tool[5] Artificial server-client, focuses on response time.

From lack of time, we were only able to try out a couple of these testing systems. None of them worked

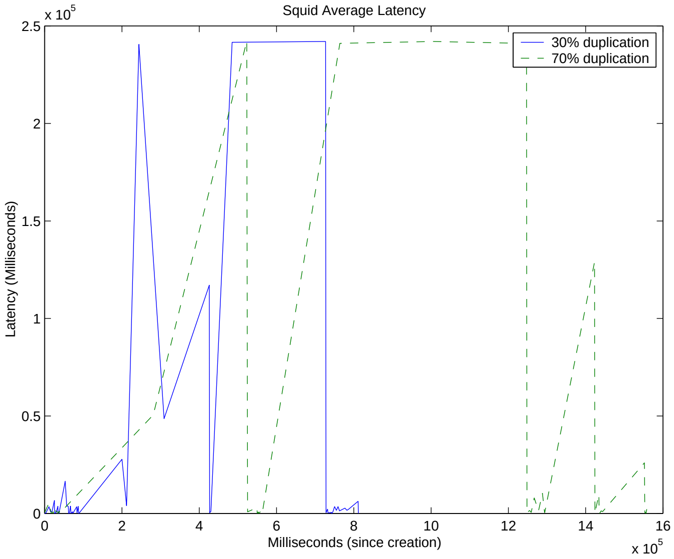


Fig. 4. Squid Average Latency

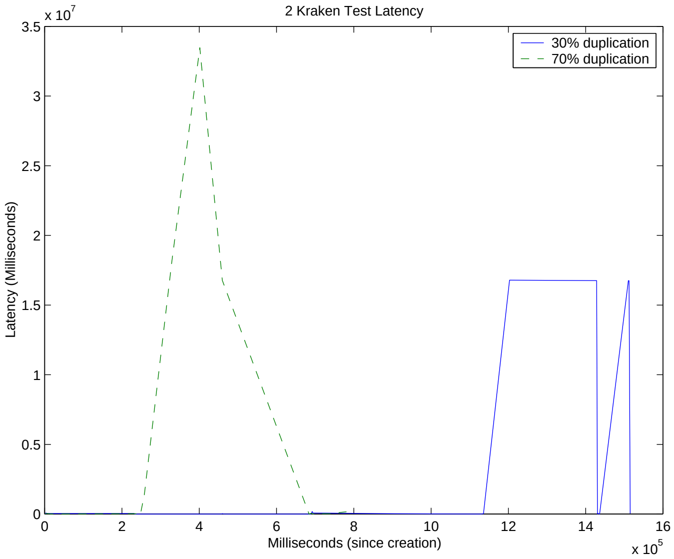


Fig. 5. Kraken Average Latency

[3] Some might say crippled.

[4] Scalable URL Reference Generator

quite right. SURGE[5] required some hacking to get it to compile. It failed to shutdown cleanly (a threads issue of some sort) and so quite often would be missing part of its logs. Web Polygraph[6] appears to have some strange time-zone misconfiguration, causing the logs to be off by exactly 60 minutes, which makes a number of functions unhappy.

Even with all these interesting tools, testing the situations we wish to try is remarkably difficult. In particular, we believe that Kraken has less degradation as the number of users increases compared to singular caches such as squid. However, this requires causing enough traffic to cause these systems to degrade. In the case of squid, the system is so optimized, it was extremely difficult to get it to degrade visibly. Keeping track of the total number of concurrent connections is hard to keep track of in both systems.

VII. Conclusion

In our initial analysis of this project, we believed that the most complex aspect would be the cache implementation. In actuality, it was the manager election scheme that gave us the most difficulties. This stems from the fact that given a minimum number of assumptions, it is really hard to build a decentralized election system that does not explode or fail to reresolve quickly. Amusingly enough, the standard algorithms mentioned in Distributed Algorithms[6] do not solve this problem.

The election algorithm presented in this paper does not actually solve the problem for all possible cases. In one particular case, a pending election in unable to be restarted by the new manager, resulting in two disjoint caches. Unfortunately, this issue is not easily solvable by the current scheme. Realizations of this form were common throughout the entire design process, reflecting the difficulty of producing a workable distributed algorithm. We believe that the real solution involves acting on election messages independent of the resolution of the current election. This removes a large amount of complexity from the election algorithm, but increases the potential number of messages sent during an election. Unfortunately, the simplicity of algorithm reflects the difficulty of proving its correctness (termination, in this case). The proof of the new scheme's correctness had not been completed by the time this paper was written.

We discovered that it is infeasible to design, implement, and validate a new distributed web cache in 4 weeks. The design and implementation took most of the time. We thought that validation and testing would be relatively straightforward, however we discovered that it is currently a hot topic of research in the field. Lastly, writing the paper brought out a number of issues that we hadn't thought of before.

A. Future Uses

The idea of an internally managed cooperative cache can be applied to solve dynamic programming problems on a large scale. Dynamic programming is an optimization that involves breaking a larger problem into a set of sub-problems, and maximizing the sub-problems to find a maximal overall solution. When applied recursively, a naive solution recomputes many intermediate subproblems. Dynamic Programming takes advantage of this by storing the results of computing the sub-problems in a reusable manner. Thus, the overall computation takes much less effort. Canonical examples of dynamic programming include matrix multiplication and line breaking. A cache can be used to store the results of sub-problems. Thus, a distributed cache can help parallelize a large dynamic programming problem. Instead of serving copies of the original data, the clients first process the data and then serve the results.

Imagine that a large data set is stored on a central server. When a client needs to perform some calculation on a subsection of the dataset, it sends a query to the manager with the data ranges and the operation necessary. The manager directs it to a cache if someone has previously computed the result. In this way, requests are modified to contain both the data needed and the computation that will be performed.

One potential application that fits these criteria is work on the genome project[15]. It is essentially one huge, sophisticated matching problem, and matching problems are good candidates for dynamic programming[17]. Specifically, many attempts will be made to match similar sequences against the genome. By catching these computations and directing them to someone who has already computed the result, much time can be saved. The genome also consists of a large amount of data, such as might benefit from caching. Lastly, many people are working on genome in parallel, mostly looking for different things, but still performing similar computation to extract the useful bits.

The power that the Kraken concept brings to this application is the ability of clients to join and leave the computation seamlessly, and the lack of centralized administration. If Kraken is being used to share pro-

[5] Pre-release Version 1.00a

[6] Version 2.5.4

cessing power between organizations, its lack of centralized management prevents one organization from officially being in control of the cache, and thus owning the results of the computation. The concept of using a work-cache to make large dynamic programming problems tractable by using an internally managed, self-healing cache is an attractive one.

Acknowledgments

The authors would like to acknowledge the suggestions and motivations of Professors Frans Kaashoek and Robert Morris.

References

[1] Akamai, http://www.akamai.com

[2] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System"

[3] R. Dingledine, "The Free Haven Project: Design and Deployment of an Anonymous Secure Data Haven" M.Eng. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2000). `http://www.freehaven.net/`

[4] Syam Gadde, Jeff Chase, "Directory Structures for Scalable Internet Caches" Dept of Computer Science, Duke University. `http://www.cs.duke.edu/ari/cisi/dirstructs`

[5] CacheFlow, `http://www.cacheflow.com/technology/tool/`

[6] Nancy A. Lynch, Distributed Algorithms p.495-497

[7] ICP and the Squid Web Cache, Duane Wessels and K Claffy, IEEE Journal on Selected Areas in Communication, April 1998, Vol 16, #3, pages 345-357 `http://www.squid-cache.org`

[8] Internet Cache Protocol, Request For Comments: 2187

[9] Microsoft Proxy Server (Cache Array Routing Protocol), `http://www.microsoft.com/proxy/guide/CarpWP.asp`

[10] S. Gribble, E. Brewer, J. Hellerstein, D. Culler, "Scalable, Distributed Data Structures for Internet Service Construction" OSDI 2000

[11] D. Karker, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins. Y. Yerushalmi, "Web Caching with Consistant Hashing" WWW8 Conference 1999

[12] L. Fan, P. Cao, J. Almeida and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol" IEEE/ACM Trans. Networking 8, 3 (June, 2000), Pages 281 - 293

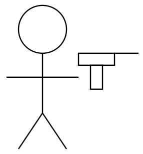
[13] Web Polygraph, `http://polygraph.ircache.net/slides/`

[14] SURGE: Scalable URL Reference Generator, `http://cs-people.bu.edu/barford/`

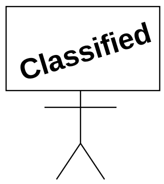
[15] E. Lander "Human Genomic DNA" Journal of Molecular Biology. v268 pp78-94

[16] Anderson, Dahlin, Neefe, Patterson, Roselli, Wang "Serverless Network File Systems", SOSP 1995, and ACM ToCS 1996.

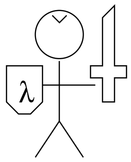
[17] Dynamic Programming Example Eric C. Rouchka, Computational Molecular Biology, BL5495 at U. Wash at St. Louis `http://www.ibc.wustl.edu/CMB/bio5495/dynamic/dynamic.html`



Joe Foley is a PhD Graduate Student in Mechanical Engineering at the MIT Auto-ID Center. He spends his free time collecting guns and pop-tarts to "prepare for the coming war." His teammates are not sure what this means but humor him anyway. He's currently working on modifying his Ford Ranger Pickup into a Mobile Weapons Platform. We wish him the best of luck and know where to hide after the fall of Civilization.



Amy Vandiver is a Masters Graduate Student in MIT Computer Science and Electrical Engineering. She is a completely well-adjusted individual, at least at first glance. She likes rocks. Amy is currently working deep-cover operations for the AeroAstro department in the Black Mesa high-particle physics research division. Don't ask. Won't tell. Dad doesn't know.



Ben Vandiver is a Masters Graduate Student in MIT Computer Science and Electrical Engineering. He is Knight of the order of Lambda Calculus. He spends his waking hours in the 6.001 lab (35-501) torturing undergrads with Scheme. There he also carefully plans the downfall of the Civilized world. Unfortunately, his method involves Java, and so it will be a while before the system is done compiling, let alone running. He spends his sleeping hours four feet above his sheets.