Semantics of Caching with SPOCA: A Stateless, Proportional, Optimally-Consistent Addressing Algorithm

Ashish Chawla, Benjamin Reed, Karl Juhnke[†][*] Ghousuddin Syed
{*achawla, breed, gsyed*}*@yahoo-inc.com*, [†]*yangfuli@yahoo.com*
Yahoo! Inc

Abstract

A key measure for the success of a Content Delivery Network is controlling cost of the infrastructure required to serve content to its end users. In this paper, we take a closer look at how Yahoo! efficiently serves millions of videos from its video library. A significant portion of this video library consists of a large number of relatively unpopular user-generated content and a small set of popular videos that changes over time.

Yahoo!'s initial architecture to handle the distribution of videos to Internet clients used shared storage to hold the videos and a hardware load balancer to handle failures and balance the load across the front-end server that did the actual transfers to the clients. The front-end servers used both their memory and hard drives as caches for the content they served. We found that this simple architecture did not use the front-end server caches effectively.

We were able to improve our front-end caching while still being able to tolerate faults, gracefully handle the addition and removal of servers, and take advantage of geographic locality when serving content. We describe our solution, called SPOCA (Stateless, Proportional, Optimally-Consistent Addressing), which reduce disk cache misses from 5% to less than 1%, and increase memory cache hits from 45% to 80% and thereby resulting in the overall cache hits from 95% to 99.6%. Unlike other consistent addressing mechanisms, SPOCA facilitates nearly-optimal load balancing.

1 Introduction

Serving videos is an I/O intensive task. Videos are larger than other media, such as web pages and photos, which not only puts a strain on our network infrastructure, but also requires lots of storage.

[*]This work was done when the author was employed at Yahoo! Inc.

Our clients access videos using web browsers. They connect to front-end servers which serve the video content. The front-end servers cache content, but are not the permanent content repository. Videos are stored in a storage farm that is made up of network attached storage accessible by all front-end servers.

To further complicate things, the video content is spread around the world. So, when a client requests content that is non-local, we must decide whether to have the client pull from the remote cluster that has the content, or copy the content from the remote cluster and serve it locally.

Video delivery is fastest and causes the least amount of load on the rest of the infrastructure if the content is cached in the memory of a front-end server. If the content must be accessed from the disk of the front-end server, the load on the front-end server increases slightly. It causes significantly more load and slower delivery if the content must be retrieved from the storage farm. Increased load on the serving infrastructure translates into higher cost to upgrade networking components and to add more servers and disk drives in the storage farm to increase the number of operations per second that it can handle. Thus, good caching at the front-end servers is important to latency, throughput, and the bottom line.

The Yahoo! Video Platform has a library of over 20,000,000 video assets. From this library, end users make about 30,000,000 requests per day for over 800,000 unique videos, which creates a low ratio of total requests to unique requests. Also, because videos are large, a typical front-end server can hold only 500 unique videos in memory and 100,000 unique videos on disk. The low ratio of total/unique requests combined with the large size of video files make it difficult to achieve a high percentage of cache hits.

A straightforward architecture, shown in Figure 1, uses a VIP (Virtual IP) load balancer which distributes requests in a round-robin fashion among a cluster of front-end servers. The VIP exposes an IP address that

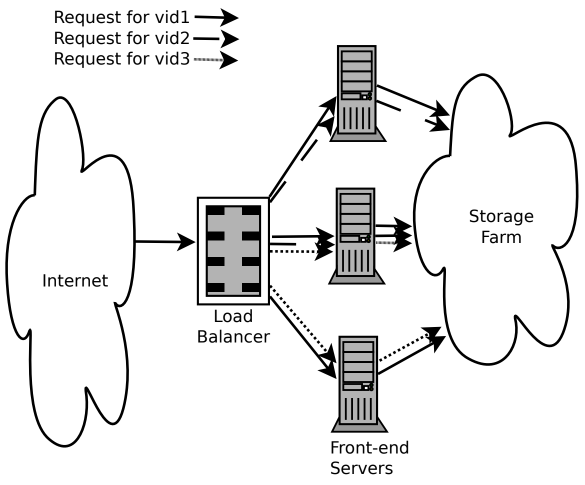

Figure 1: A straightforward content serving architecture using a hardware load balancer VIP (Virtual IP) with front-end server that are connected to a shared storage farm.

the clients connect to. The VIP routes connections to front-end servers to balance load and mask failures of front-end servers. Front-end servers manage their cache by promoting every requested item while demoting the least recently used item.

This was our initial content serving architecture that we used in production. Unfortunately, this straightforward approach results in more requests to the storage farm, due to cache misses at the front-end servers, than the farm can handle. Our storage farm has copious space but limited bandwidth. The front-end server disks are a secondary bottleneck because memory cache misses exhaust the disk throughput before the CPU of the front-end server can be fully utilized.

In a cluster of front-end servers behind a VIP, each piece of popular content will end up cached on multiple servers. For example, in Figure 1 `vid1` is a popular video that ends up cached on all the front-end servers. `vid2` and `vid3` may have only been requested twice each, but that resulted in each video being cached on two nodes.

This is grossly inefficient compared to caching each piece of content on only one server and routing all requests for that content to the server where it is cached. If the cluster's collective cache stores as few copies of each video as possible, then it will be able to store as many unique videos as possible, which in turn will drive down cache misses as low as possible. Eliminating redundant caching of content also reduces the load on the storage farm. An intelligent request-routing policy can produce far more caching efficiency than even a perfect cache promotion policy that must labor under random request routing.

Maximizing caching efficiency via request routing introduces practical challenges. It is difficult to keep a request router's knowledge in synch with the actual cache of each front-end server. Furthermore, even if a request router has a real-time database of cache contents, a database lookup on every user request is a non-trivial latency. Also, most deployments must have more than one request router, which raises the possibility of two different routers independently making a different decision of where to place new content that is not yet in cache, or where to re-locate content when a front-end server leaves or enters the cluster.

The cache promotion algorithm is a natural place to look for improvement. A better promotion policy offers us significantly more memory cache hits and is therefore a step toward relieving the disk throughput bottleneck. The accesses to the storage farm, however, are reduced only marginally by a good promotion policy, because the disk cache miss percentage is low to start with. In some circumstances, delaying a page-in from permanent storage to the disk cache until we are confident that a piece of content is promotion-worthy actually results in more disk cache misses than automatic promotion does. Therefore, another solution is necessary.

Of course the above discussion does not consider the problems arising from the geographic distribution of the content. Not all content is available at all locations. The cluster of servers closest to the user, *nearest locale*, may not be the cluster storing the content, *home locale*. So when we put together the caching discussion and the geolocality, we end up with the following possible user experiences:

1. nearest locale and cached ⇒ excellent experience
2. nearest locale and not cached ⇒ average experience
3. home locale and cached ⇒ average experience
4. home locale and not cached ⇒ below average experience

To get excellent user experience for the most users we need to be able to cache popular remote content locally.

In this paper we describe SPOCA, a system for consistent request routing, and Zebra, a system for routing requests geographically. Both systems have been in production for a few years now at Yahoo! The contributions of this work are:

- We describe a system that is actually used in production in a global scenario for web-scale load.
- We show the real world improvements we saw over the simple off-the-shelf solution in terms of performance, management consolidation, and deployment flexibility.
- SPOCA implements load balancing, fault tolerance, popular content handling, and efficient cache utilization with a single simple mechanism.
- Zebra handles geographic locality in a way that fits

nicely with the mechanism used locally in the data centers.

- We are able to implement all the above with only soft state.

2 Requirements

Our content distribution network is faced with different traffic profiles for its various delivery modes. To handle this, profiles are divided by types of content into pools. This allowed us to adjust the provisioning and policies of the pools to accommodate the traffic profiles. The three main content pools are: Download pools (DLOD), Flash Media Pools(FLOD), Windows Media Pools(WMOD). FLOD is made up of a relatively small library of files and a high average popularity; for DLOD the traffic consists mostly of a large number of unpopular files; and WMOD streaming must deal with both a huge library and some very hot streams. A high level goal for the platform was to merge these pools and be able to manage the diverse requirements of the different traffic profiles in an adaptive way.

A naïve approach to partitioning a pool among a group of front-end servers would be to maintain a catalog that associates each content file with a particular front-end server. The catalog approach is, however, impractical because the set of servers in a location is constantly changing. It would be too time-consuming to re-index the entire content library every time a new front-end server became available, or a current server became unavailable, or a former server re-entered the pool after having been temporarily unavailable. Therefore Yahoo! uses a stateless addressing approach.

For stateless addressing, the inputs are a filename and a list of currently available servers; the output is a server from the list. This eliminates the need to maintain and communicate a catalog. The tradeoff is that Request Router must recalculate the destination server on every request, but fortunately the cost is only microseconds in practice. The same input always produces the same output, so two different Request Router servers, without communicating to each other, will address the same file to the same front-end server within a pool.

We have additional stringent requirements for our addressing algorithm. First, it must partition the set of filenames proportional to different weights for different front-end servers in heterogeneous pools. For example, a newer server might have twice the capacity of an older server, and therefore should serve twice as large a portion of the content library.

The proportionality requirement rules out the use of a distance-based consistent hashing algorithm, although such algorithms are consistent and stateless. Such an algorithm assigns an address to each front-end server, and

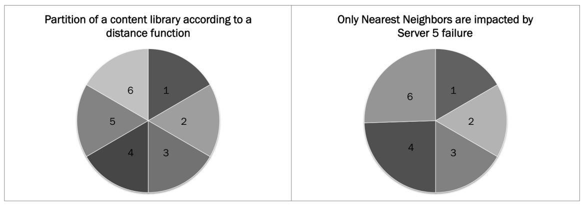

Figure 2: Distance-based Consistent Hashing

assigns a file to the server with the address closest to the hash of the filename, according to some distance function. To see that distance-based consistent hashing does not respect server weights, suppose to the contrary that some configuration of front-end server addresses did respect server weights. Suppose that some server then became unavailable. All of the content for which the unavailable server had been responsible would fall to its nearest neighbors by the distance function. Servers farther away by the distance function would pick up none of the load. Therefore, the content library would no longer be partitioned proportional to server weights (see Figure 2).

In some circumstances it might be reasonable to permit the load of a missing server to be redistributed to its nearest neighbors only. For Yahoo!, however, one reason for a server to be missing is that the server was overloaded. If its load then falls entirely on a few neighbors by distance, they may also become overloaded and fail as well, creating a domino effect that brings down the entire pool. Therefore it is critical that whenever a server leaves the pool, its load is distributed among all remaining servers, proportional to their respective weights. (Figure 3 show an acceptable redistribution.)

Our second requirement for our addressing algorithm is that it be optimally consistent in the following sense: when an front-end server leaves or enters the pool, as few files as possible are re-addressed to different servers, so that caching is disrupted as little as possible.

The two requirements of respecting weight and re-addressing a minimum number of files are quite limiting. For example, suppose that a pool has three front-end servers of weight 100, and two servers of weight 200. If a new server of weight 200 is added to the pool, not only must the new server be assigned two-ninths of the files in the content library as a whole, but more specifically for each of the other servers it must take over two-ninths of the files that server was handling.

Figure 3 depict roughly what must happen when servers join and leave the pool if weights are to be respected and a minimum number of files are to be re-assigned.

A third requirement on our hashing algorithm arises from the fact that a proportional distribution of files

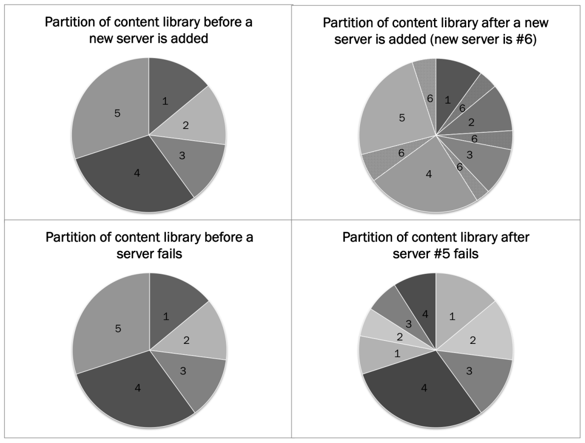

Figure 3: Proportional Redistribution of Load

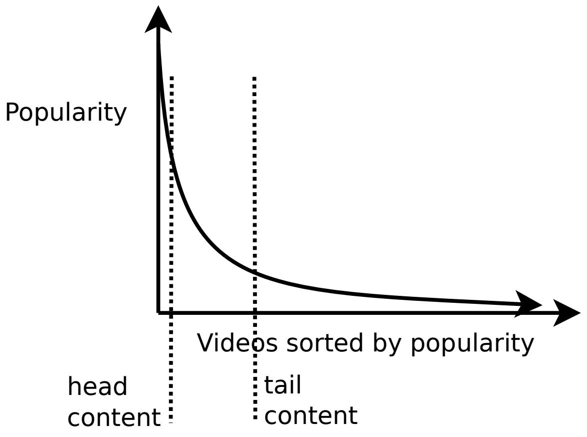

Figure 4: Video popularity follows a power law distribution.

among servers does not necessarily result in a proportional distribution of requests. Perfect caching is in conflict with perfect load balancing, because some files may be more popular than others. Indeed, a single file may be hotter than any front-end server in the pool could handle by itself. There must be a way to detect hot files and/or detect overloaded servers, so that traffic can be distributed away from affected servers.

The load-balancing requirement could be hacked in to the distribution system as an exception to our consistent hashing algorithm, for example simply by saying that files beyond some popularity threshold are evenly distributed between all front-end servers.

Instead of distributing popular streams to all front-end servers, we distribute only to two or three or however many are necessary to meet demand. The means that our algorithm must produce more than a consistent server as output; it must produce a consistent server list.

Figure 4 shows distribution of requests over the content served by our video service. Most requests are for a small number of popular or head content; however, there are still many requests spread over the long tail of less popular content. Similar distributions of popular content have been observed for other services as well [18].

The head content served by Yahoo! is so popular that a single front-end server cannot handle all user requests for a single video. The request router must have a mechanism to distribute hot content to more servers than one front-end server. A few extra cache misses are less problematic than a server being completely overwhelmed by a piece of head content.

The majority of user requests for video, however, are requests that could (and should) be handled by a single server. Therefore, it is equally important that the number of pieces of tail content addressed to each server be proportional to that server's capacity. Although our front-end servers are generally homogeneous, new front-end servers we add to the cluster inevitably have different capacity than existing servers. As we distribute requests across the cluster, we need to take into account the different capacities of the front-end servers.

Our platform is deployed as a cluster of video servers spread across the world, so we need to take geolocality into account. However for unpopular content, it is more effective to serve the content from a remote location rather than try to make the content geographically local. As content becomes more popular we want clients to access them from servers that are close to them.

Finally, our video service is a 24/7/365 operation. We need it to be elastic: we need to be able to add and remove servers from the running system. We also need it to be fault tolerant: we need to gracefully handle the failure and recovery of front-end servers.

3 Overview

We drive caching and locality decisions based on content popularity. Zebra decides which non-local popular content should be cached closer to the requestor. Local content will be cached at an optimal number of local servers based on popularity.

Zebra routes popular requests for popular content to the cluster closest to the nearest locale and unpopular content to the home locale. Zebra initially considers all content as unpopular, so the first request for a particular video will be directed to the home locale. Subsequent requests for the same content will cause Zebra to consider the content as popular and direct requests to the nearest locale. Because of the number of videos served we want to do this popularity detection in a way that uses only soft state and can be tracked with a fixed, and relatively small, amount of memory.

Local content caching uses a Stateless, Proportional, Optimally-Consistent Addressing Algorithm (SPOCA) to route content requests to our front-end servers rather than simple VIPs. Given the name of a piece of content and a list of front-end servers, the algorithm deterministically maps the content to a single front-end server. Two request routers will independently arrive at the same mapping, and the same request router will make the same mapping repeatedly without having to remember anything. The computation is faster than database lookups, and the only communication overhead required is the list of active front-end servers.

SPOCA is consistent beyond being deterministic. Front-end servers will occasionally drop out of the cluster due to outages or maintenance, and new servers will occasionally be added to refresh technology or increase capacity. When the list of active servers changes, it is unacceptable for the mapping of content to servers to wholly change. Indeed, for optimal consistency, content should never be re-mapped from server A to server B unless A left the cluster or B joined the cluster. In other words, if server A and server B are present both before and after the cluster changes, then no content can be remapped from one to the other.

To serve content we assign each piece of content served a unique name the form `Content-ID.domain`. The `Content-ID` is deterministically derived from the identifier of the content, the hash of the filename for example. `domain` corresponds to the pool to which a piece of content belongs. `domain` also represents a valid DNS subdomain managed by Yahoo!. Thus, `Content-ID.domain` is a valid DNS name that can be resolved by a DNS server. We have a special DNS server, called Polaris, that works with Zebra and SPOCA to route a request to the appropriate server.

During DNS resolution Zebra and SPOCA determine the front-end server to route the request to; Polaris directs the client to that server by resolving the request for `Content-ID.domain` to the determined front-end server's IP. Web pages that embed content to be served uses a URL of the form `http://Content-ID.domain` to address the content.

4 The Zebra Algorithm

Zebra handles the geographic component of request routing and content caching. Its main caching task is to decision out when requests should be routed to content's home locale and when the content should be cached in the nearest locale. Zebra makes this decision based on content popularity.

Zebra tracks popularity using soft state with a limited amount of memory. Bloom filters [2] seem like a good data structure to use for this kind of tracking. As requests for content come in, we can add them to the bloom filter to track popularity. Unfortunately, it is not possible to remove content from the bloom filter. So we need a way to stop tracking content that is no longer popular.

Rather than using a single bloom filter, Zebra uses a sequence of bloom filters (we use 17 filters in production). Each bloom filter represents requests for a given interval, on the order of hours. We keep a fixed number of filters in the sequence and expire older filters as new are added. Content is considered popular based on the union of the intervals. We optimize the popularity check by combining all the bloom filters older than the current interval into one after the start of a new interval, so that popularity checks involve lookups in only two bloom filters.

Note that even if we had used a more sophisticated bloom filter structure, such as counting bloom filters [7], we would still need to track entries to be deleted because they are no longer popular. Our strategy of using a sequence of bloom filters both tracks and removes entries that are no longer popular using simple bloom filters.

If content is deemed popular, it is in one of the two bloom filters, the content will be cached locally. Not only does Zebra enable more effective geographic caching, it also enabled us to decouple delivery from storage. It now makes sense to have content serving front-ends in a data center that has no content storage servers. Popular content will be cached locally at the front-end servers and unpopular content requests will be routed to its home locale.

Zebra determines which serving clusters will handle a given request based on geolocality and popularity. SPOCA determines which front-end server within that cluster will cache and serve the request.

5 The SPOCA Algorithm

To maximize the cache utilization at the front-end servers and thereby minimize the load on our storage farm SPOCA aims to localize requests for a given video at a single server. This will allow the best utilization of the aggregate memory of the front-end servers. We also need to balance the load across the front-end servers and handle failures and server additions. Requests for a popular video may overload a single server, so we need to be able to assign the handling of such content to multiple front-end servers. Finally, we have serving clusters around the world, so we need to take geolocality into account.

Figure 5 illustrates how we would like content to be served. Each video is served by one server. This decreases the load on the storage farm and it more effectively uses the cache of the front-end servers. Since `vid1` is a popular video we would like it to be served by multiple servers.

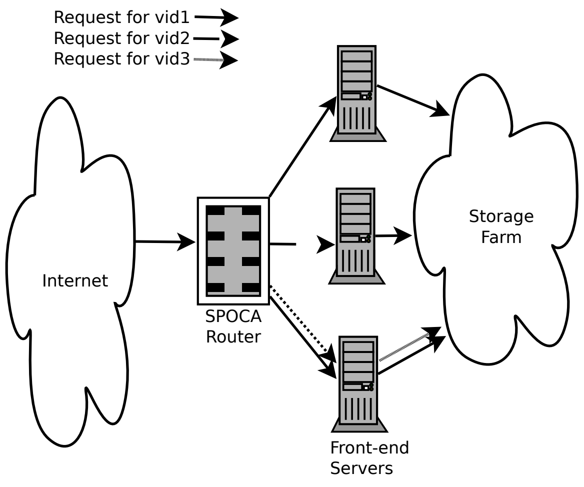

Figure 5: SPOCA consistently routes the same video to the same front-end server. It also increases the number of servers serving popular content. (e.g. vid1)

SPOCA fulfills our requirements using a simple content to server assignment function based on a sparse hash space. Each front-end server is assigned a portion of the hash space according to its capacity.

The SPOCA routing function takes as input the name of the requested content and outputs the server that will handle the request. The SPOCA routing function uses a hash function to map names to a point in a hash space as shown in Figure 6. Each front-end server is assigned a portion of hash space proportional to its capacity. Not every point in the hash space maps to a front-end server, so when the hash of the name of a requested video maps to unassigned space, the result of the hash function is hashed again until the result lands in an assigned portion of the hash space.

Using this hashing scheme SPOCA load balances content requests using random load balancing in such a way that it can gracefully handle failures, the addition and removal of front-end servers, and popular content.

5.1 Failure handling

If a front-end server fails, the portion of the hash space that was assigned to the failed server becomes unassigned as shown in Figure 7. Requests that would have been assigned to the failed server are rehashed as normal until they land in a region assigned to an active server. This has the nice property that only the content assigned to the failed server will be re-routed to other servers in a balanced fashion. Content assigned to the servers that have not failed will continue to be served by those servers, which allows us to continue to utilize effectively the cached content at those servers.

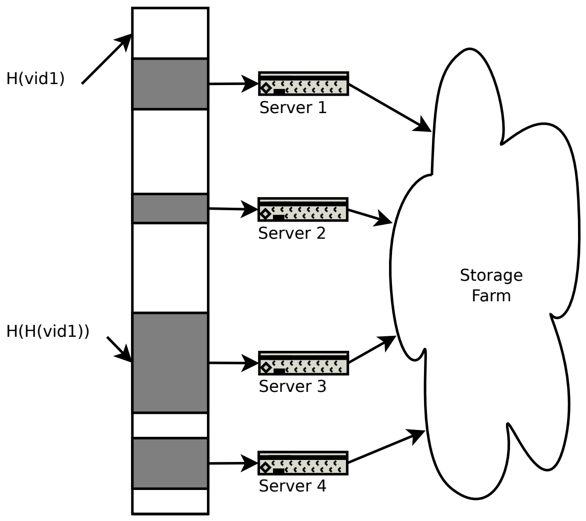

Figure 6: An example assignment of the SPOCA hash map. The name of the requested video initially hashes to empty space, but when hashed again is assigned to server 3.

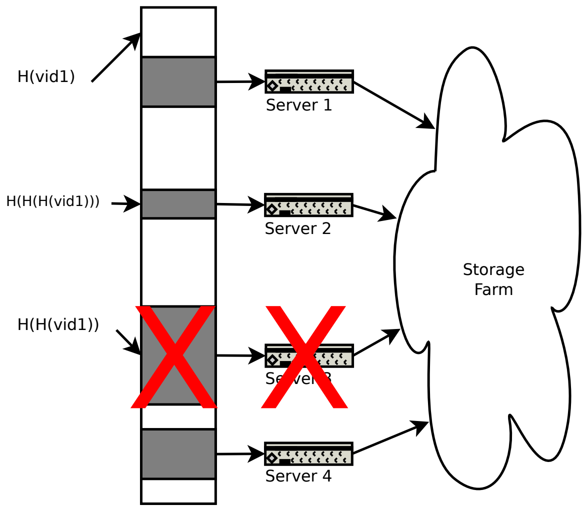

Figure 7: When server 3 fails, the content handling for the named video is reassigned to server 2.

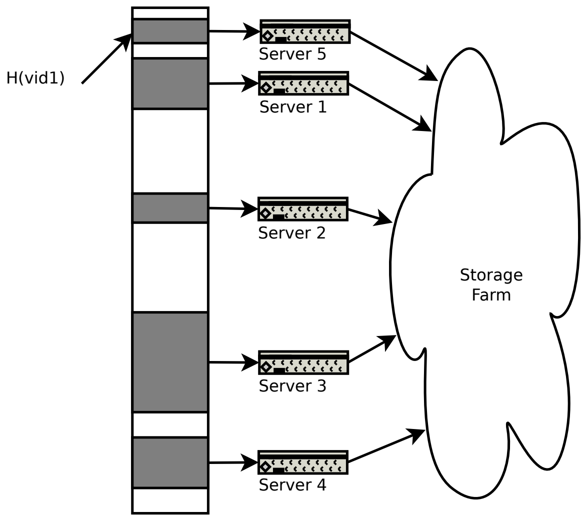

Figure 8: When server 5 is added, the content handling for the named video is reassigned to server 5.

5.2 Elasticity

New servers are mapped into the unassigned portion of the hash space as shown in Figure 8. When this happens a portion of the content assigned to other servers will now be assigned to the new server. For example, in Figure 8 the video that was previously handled by server 3 will now be handled by the new server, server 5. Server 3 may have `vid1` content in its cache because of previous requests. Eventually server 3 may end up replacing `vid1` with other content it is serving. If `vid1` becomes popular or server 5 fails, server 3 will again start serving `vid1`.

Servers are removed from service by simply removing their assignments from the hash space. This will cause the mechanism described in the previous section to kick in and content served by the removed server will be spread to other active servers.

5.3 Popular content

SPOCA tries to minimize the number of servers that cache a particular piece of content to maximize the aggregate number of cached objects across the front-end servers. This strategy works well with tail content, unpopular content, but it can cause front-end servers for head content, popular content, to become overloaded. So, for head content we need to route requests to multiple front-end servers. We do this routing using a simple extension to the SPOCA routing mechanism described earlier. Popular files are handled by the same algorithm that deals with missing servers, so a page-in for either may benefit the other.

To handle popular content, the request router stores the

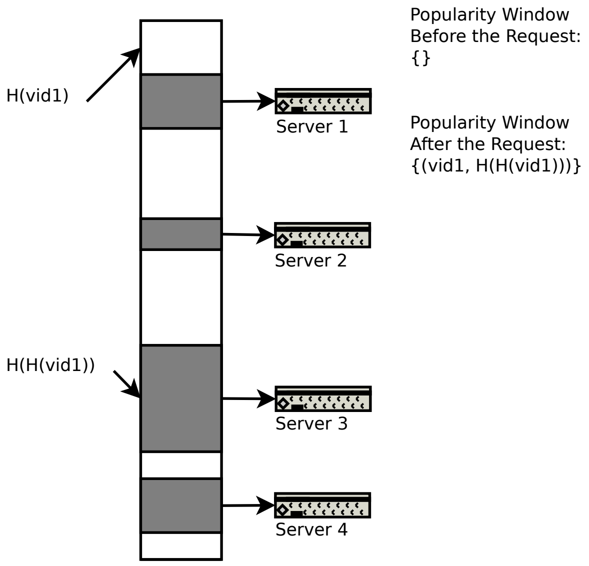

Figure 9: When a request for `vid1` is received, SPOCA routes the request to server 3 and stores the hash in the popularity window.

hashed address of any requested content for a brief popularity window, 150 seconds in our case. On every new request, the request router checks whether it has a saved hash for the requested content. If no hash is present, the request is routed to a front-end server using the normal procedure. If a hash is present, meaning the content has been served within the popularity window, routing will start using the stored hash rather than the name of the requested content. In either case, only the final hash (i.e. the address where the request was ultimately routed) is saved along with the content name.

Figure 9 shows the routing for `vid1` with the popularity window. Because popularity window did not have a entry for `vid1` the request will be routed the same as Figure 6. If another request is received in the popularity window, as shown in Figure 10, the routing will start with the hash stored in the popularity window rather than the hash of `vid1`. Note that the server that handles overflow is the same server as handles requests for `vid1` if server 3 fails as shown in Figure 7.

When the popularity window expires, the stored hash for each object is discarded regardless of how recently it was used. It follows that each object may be mapped to as many different servers as there are requests for that file within the popularity window. For a given object, the sequence of servers to which it is routed is the same in each popularity window, with the number of requests determining how deeply into the sequence the mapping advances. An object which is never requested twice within the same popularity window is always mapped to the same server.

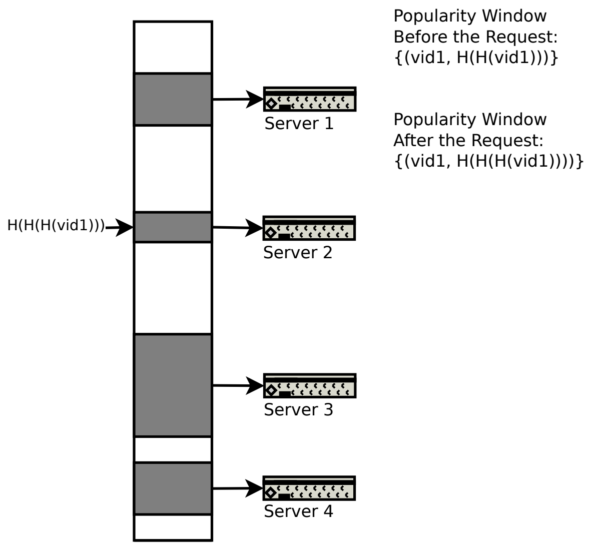

Figure 10: When the next request for `vid1` is received, SPOCA uses the hash in the popularity window to start the routing and routes to server 2 and stores the updated hash in the popularity window.

If a file is temporarily popular enough to be distributed to two servers, both will cache it. If later the first server is unavailable, the second server will resume primary responsibility for the file it already has cached, rather than that file being re-assigned elsewhere. Similarly if a server goes down for a while, the files for which it is responsible will be cached elsewhere. If, after that server has come back on line, one of its files becomes popular enough to be distributed to a second server, that second server will already have the file in cache.

The load balancing for hot files is not absolutely stateless, but the request router does not need to store an entry for each of the 20,000,000 unique files in the library, or even the 800,000 unique files requested in a day, merely the 20,000 unique files it has seen in the last 150 seconds. This amount of data can easily be held in memory.

The larger the time window, the greater the permitted load imbalance between servers. The shorter the time window, the greater the unnecessary duplication of files. It is a balance between inefficiencies.

Keeping a histogram of recent files would be greater overhead, and would in any case not answer the important question, namely whether we prefer the inefficiency of load imbalance or the inefficiency of redundant copies. The information necessary to decide about that tradeoff is not available to the stream router.

5.4 Memory management

The original media server caching policy waited too long to bring content from the filers into local cache. We decided to use a more aggressive policy with SPOCA. When explaining this policy we use the terms page-in and page-out to describe the actions of populating the cache with a piece of content and evicting a piece of content from the cache. The new caching policy, embodied in SPOCA, goes to the extreme to correct the problem of unresponsiveness: SPOCA calls for content to be paged into local cache as soon as it is requested. This raises an obvious question: is immediate page-in the right approach?

The traditional concern with aggressive caching is that it causes churn, which is to say that a less-popular item will be brought into cache, forcing a more-popular item to be deleted from cache. The old caching system reflected the traditional mindset: that system was designed to prevent churn by tracking whether an item was truly popular before paging it in. In our configuration, however, churn is not the primary issue.

In light of increasing disk sizes and improvements to the distribution of requests among media servers, it is reasonable to suppose that a media server can cache every stream that is requested for an entire week. We observed that whenever a file is requested, the probability that it will be requested again within a week is 69% for ads, 80% for audio, and 83% for video, so even a random new file we are getting our first request for is likely to be more popular than the oldest file in our cache.

However, even if churn is low, even if cache misses have been reduced to an absolute minimum, there is another potential reason not to page in aggressively, namely that paging in itself causes load on the filer. If we recall that the objective of the caching system is to reduce filer load, then we must reduce both cache misses and page-ins. The policy of immediate page-in may save less in cache misses than it costs in page-ins.

Indeed, an average page-in may place a greater burden on a filer than an average cache miss, because in the case of a cache miss, the user may not view the entire stream, so the filer can quit serving it partway. We have a question that cannot be decided in the abstract: It takes real data to know whether 20% of the average stream is viewed, or more, or less.

It is possible, however, that a page-in places a lesser load on a filer than a cache miss. This is because the filer can serve a page-in request at full speed, reading the whole file contiguously, whereas to stream a file the filer has to stream it out byte by byte.

What we observed was that for the video media servers, the immediate page-in policy is correct. In most cases we load the filer the least by paging in immediately, which works very well with our general desire to be as responsive as possible.

5.5 SPOCA Implementation

SPOCA's consistent hashing algorithm implementation is based on the standard C pseudo-random number generator. In extreme circumstances, linear congruence generators may have undesirable properties, but for distributing traffic based on filename they are quite sufficient.

Since each file has a `Content-ID` this number is used as the *seed* for our pseudo-random number generator. Thus it can generate an arbitrarily-long, deterministic sequence of numbers, uniformly scattered in the unit interval.

When an front-end server is entered into a pool, it is assigned a segment of the unit interval that does not overlap with any other server's segment. The length of the segment represents that server's weight. Upon receiving a request for a file, SPOCA generates pseudo-random numbers within the unit interval until one lies within a segment that has been mapped to a server. Algorithm 1 shows the basic logic to map a request for content, `filename`, to a server. At the heart of the algorithm is the function `maptoserver(seed)`, which returns the server whose assigned segment includes `seed`. Failures are also reflected in `maptoserver(seed)`. If SPOCA detects that a front-end server has failed, `maptoserver(seed)` will return `null` for any `seed` that falls in the failed servers assigned interval.

When a new server is added to the pool, the load is evenly distributed among all the servers. What it means is that the new server takes some load from each of the existing servers in the pool. On the same lines, when a server goes down, it takes the load which that server was handling and re-distributes to the other servers in the pool.

Algorithm 1 Pseudo-Random Generation Algorithm

```
1: seed := filename;
2: seed := rand(seed);
3: while maptoserver(seed) = NULL do
4:     seed := rand(seed);
5: end while
6: return maptoserver(seed)
```

In order to allow for the painless addition of future front-end servers, our servers typically cover only 1% to 25% of the unit interval, depending on our anticipation of future growth. Note that if the mapping of front-end servers covers only 1% of the unit interval, then SPOCA will have to generate an average of 100 pseudo-random numbers to distribute one request. Even so this algorithm causes negligible latency, because linear congruence generators are so simple.

On the rare occasion that one wishes to add servers after an interval becomes entirely mapped, it is best to shrink all existing segments to a some fraction of their size, causing a corresponding fraction of disruptions in caching. However, sparsely assigning the unit interval, creates enough room for years of foreseeable needs. By starting with only 1% of the unit interval assigned, we can grow a cluster to almost 100 times its initial size before worrying about running out of room. In our experience so far, we have been able to plan ahead and avoid any such re-mapping.

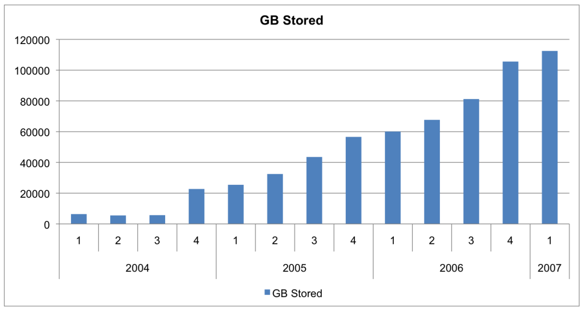

Figure 11: Data Growth

SPOCA controls distribution of popular files by saving a `seed` for each `filename` for a configurable length of time T. Every time T elapses, all saved seeds are thrown away. If a request arrives for a file for which the request router has a saved `seed`, that file is deemed to be popular, and the generation of pseudo-random numbers starts from the saved `seed` rather than from the `filename`. Algorithm 2 shows how the previous algorithm can be modified to incorporate this behavior.

Algorithm 2 Distribution Algorithm

```
1: if savedseed[filename] = NULL then
2:     seed := filename;
3: else
4:     seed := savedseed[filename];
5: end if
6: while maptoserver(seed) = NULL do
7:     seed := rand(seed);
8: end while
9: savedseed[filename] := seed
10: return maptoserver(seed)
```

A stream requested N times within the interval T may be distributed to as many as N servers. Every time the saved seeds are thrown away, the pseudo-random sequence starts over from the filename. If a file is never popular enough to be requested twice within T, the filename will always be used as the seed, and thus it will always be distributed to the same server.

6 Evaluation

We use historical data we have collected over time from production to evaluate the quality and performance of our proposed algorithms. The dataset shown in Figure 11 covers Q1 2004 to half of Q1 2007. The amount of content stored and distributed has been approximately doubling year-over-year.

When storage and distribution each double, requests served from the filers quadruple. We could run into a situation where we would need four times the filers to support scaling delivery by two times. The reason for this is because the cost model was not linear. There were limitations on IO performance of the filers and it was not a linear growth. We were consistently seeing 100% CPU hit on the filers even for the 10% cache miss. Without SPOCA, we would need more hardware as we start serving more requests from the filer.

We observed in our production environment that load balancing with SPOCA is three times better than load balancing by VIP because SPOCA's hashing function deterministically routes to the right server to serve the request whereas VIP routing does simple random routing without regard to where content may be cached. Note that this is not one server getting three times the load of another, as might happen in a peer-to-peer system which does not guarantee a partition of the address space proportional to server weights. The variation among servers is rather three times the small amount produced by random request routing. This increased variation is smoothed out by the law of large numbers: the more requests distributed by the request router, the closer to an average load each front-end server gets. For the delivery platform, load balancing has never been an issue since SPOCA was implemented, whereas the caching problem SPOCA solved was quite serious.

Over 99% of files accessed in any given day are not accessed often enough to trip the popularity trigger. Therefore these files are cached on only one front-end server each. As a file grows more popular it does not necessarily get distributed to the entire cluster. Instead it is spread to two, three, four servers and so on in linear proportion to how popular the file is. No more cache misses are created than are necessary for load balancing.

Only 0.01% of all files are popular enough to be cached on all front-end servers. No additional mechanism for identifying such files is necessary. Each server's probability of being the next server in the sequence to which a hot stream is mapped is proportional to that server's weight. The fail over mechanism therefore load-balances extremely popular content in the same proportions as it maps tail content.

With SPOCA we have been able to reduce cache misses by 5x. Each item is now cached on as few media servers as possible (usually one media server in each location instead of all servers in a pool). There is also an increase in stability with larger globally distributed clusters of front-end servers. Our serving clusters grew from 8 servers to 90. The management of this single large cluster was much easier than managing many small clusters. Because SPOCA automatically adapted to different workload profiles on a per file basis, we no longer needed to use separate pools for the different workloads. We were eventually able to consolidate 11 pools into a single pool, which also allowed us to further simplify management.

Without implementing SPOCA, video streaming from filers was approximately 219 terabytes daily. Refer to Figure 12 which shows the graphs for filer storage v/s filer streaming of video assets. After implementing

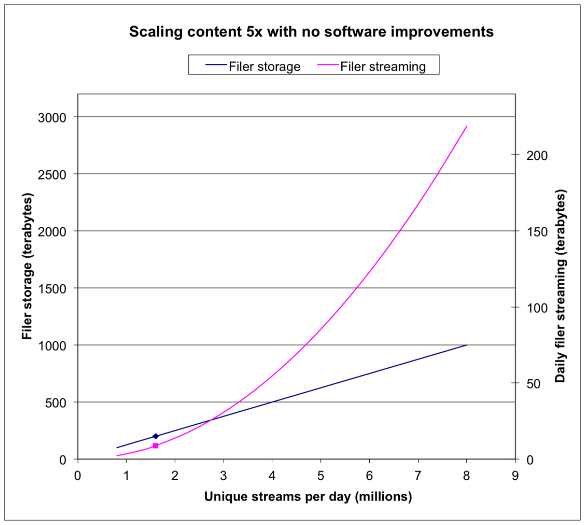

Figure 12: Without Software Improvements

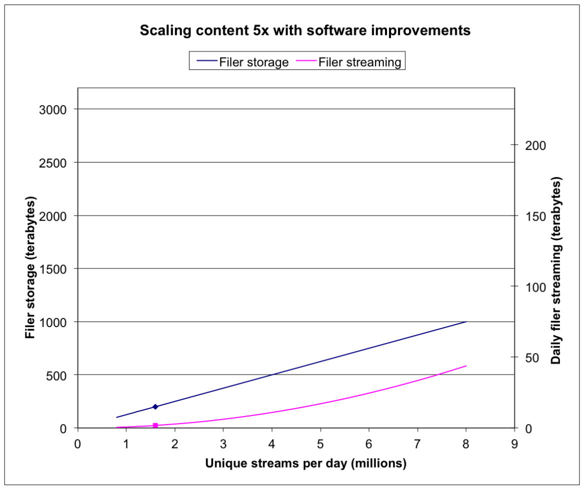

Figure 13: With Software Improvements

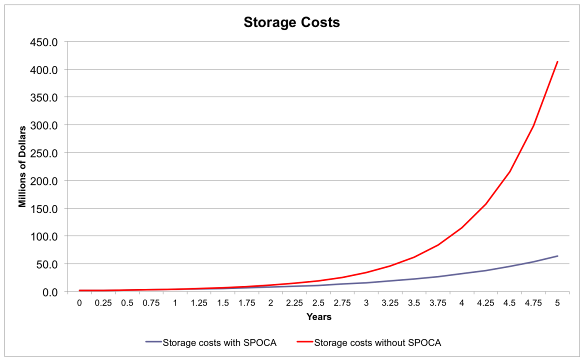

Figure 14: Storage Costs With and Without SPOCA

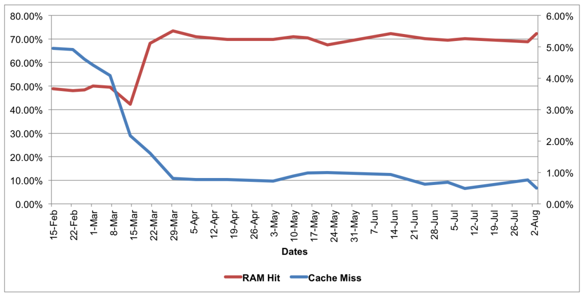

Figure 15: Increased memory cache hits across all server by lowering the popularity window from 300 secs to 240 secs. The change was rolled out to servers over a period of weeks.

SPOCA, we see a drastic reduction (5x) in the number of bytes streamed from the filers. Video Streaming from filers with SPOCA reduced to approximately 44 terabytes daily. Fig 13 shows the performance improvements with SPOCA.

A further benefit of SPOCA has been substantially increased memory cache hits. Torso content (less popular than head content but more popular than tail content) was formerly distributed between many servers. It was in disk cache everywhere, but qualified for promotion to the memory cache nowhere. With consistent addressing, however, torso content requests are collected and focused on a single front-end server, sometimes making the content popular enough for promotion to memory cache on one server before it needs to be paged into disk cache anywhere else.

As the video assets stay more in cache, streaming from filers is reduced and hence the need to add more filers also goes down as we saw from Fig 13.

Fig 14 is a projection model of the costs saving on the filer. More than $350 million dollars in unnecessary equipment (filer costs, rack space etc) alone can be saved in five-year period of running with SPOCA. In addition to the substantial savings in filer costs, the hidden cost savings included Power savings for running the equipments and data center utilizations.

The size of the popularity window is a tunable parameter. A smaller window results in fewer disk cache misses and more memory cache hits at the cost of greater load imbalance due to hot files. The window of 150 seconds for the SPOCA has driven the cache misses low enough while keeping the load balancing is even enough, such that we have not had to fiddle with the parameter to find the perfect sweet spot.

Table 1 shows us the traffic pattern for one of our data centers (S1S). It is pretty impressive that S1S missed 0.7% on the Flash workload and 0.4% on the Download workload on 3/14. The numbers would have been even better, but we had a server that lost it's RAM drive which adversely affected those numbers. The traffic pattern is a little variable, but on the whole SPOCA has apparently reduced cache misses in S1S by almost a factor of ten. From this table (Table 1), we also see that the Download RAM hit went down from 70% on 3/7 to 21% on 3/10 and to 14.2% on 3/14. This drop can be attributed to server misconfigurations or the in-memory index database getting corrupted after server reboots.

In production we measured how drastically we reduced the costs to store the bytes on the memory v/s the disk cache. Our measurements indicate that SPOCA improved memory cache hits from from 45% to 70%, and the overall cache hits increased from 95% to 99.6%. Due to this an item stays in cache for an increased duration (5X of the time it stayed earlier. e.g. what stayed in for only three hours now stays for 15 hours). We can also scale storage hardware linearly instead of quadratically, thus directly positively impacting cost.

Fig 15 shows the impact of increased memory cache hits from lowering the popularity window (referred to as T in Section 5.5) from 300 seconds to 240 seconds. This window governs the reset of the sequence of servers which are obtained from `maptoserver(seed)` function. A shorter window results more the cache hits because the requests are concentrated on fewer servers. But there is a trade-off here. If we make the window too small, we can use too few servers to serve a popular stream and overload the servers. The figure shows the three main pools in our three main locations. Further adjustments and measures have lead us to use a popularity window of 150 seconds in our production environment.

The improvement in memory cache hits from SPOCA was less dramatic: it only improved from 49% to 70%. However, the improvement in cache misses was more dramatic: it went from 5% all the way down to 0.5%, i.e. a factor of ten! We later bumped up the memory cache hits by adding more RAM, refreshing some of the older hardware with beefier boxes.

	2/26	3/1	3/5	3/7	3/10	3/14
Download cache miss	9.7%	7.2%	4.3%	3.7%	1.8%	0.4%
Download cache hit	90.3%	92.8%	95.7%	96.3%	98.2%	99.6%
Download RAM hit	42.4%	66.0%	63.4%	70.0%	21.0%	14.2%
Flash Cache miss	21.8%	13.5%	22.0%	14.8%	2.5%	0.7%
Flash RAM hit	57.2%	81.4%	66.1%	71.5%	90.0%	90.1%

Table 1: Cache Hit and Misses for the Download and Flash Pools in S1S data center

6.1 Churn Times

To amplify the hit rate, some classical caching schemes can be employed. For example, prefetching, whereby the content is cached to anticipate future requests. These techniques can also reduce the average hops between servers for content delivery. However, we cannot prefetch all the content, because of various limitations including bandwidth and storage costs. One way to evaluate the effectiveness of a cache is to look at its churn time. We define churn time as the period of time an item remains in the cache. A high churn time means that on average an item stays in cache for a long time before being replaced. There have been various models for studying the right cache size for the content type and churn times [14, 22].

We examined the churn times on our various pools of servers across a couple of different data centers. Table 2 has the statistics on churn times from disk cache and Table 3 has the churn times from memory cache from the various pools across multiple data centers. The way to read the table is this: If something is being removed from cache to make room, then it has not been accessed in X time. In table 2, we see that the churn time for DAL - DLOD pool is 8.2 days, which means that the content stays in disk cache for 8.2 days before its churned out. Similarly, in S1S data center, content would stay in Windows Media Pool (WMOD) for 2 years before its removed. However, if we look at table 3, we see that the memory churn time for DAL - DLOD pool is 2.5 hours and in S1S, churn time for Windows Media Pool is 3.8 hours.

The bigger numbers are based on our projection model because when we did this study, the system was not in production for long enough.

7 Related Work

Figure 16 shows a comparison with options engineering had during the design of SPOCA. Our main requirements at that time were proportional distribution of head and tail content, Consistent addressing/Good caching scheme and the ability to scale by adding/removing servers. Some of the schemes we looked at addressed part of our requirements but none addressed all. We had Foundry Networks [3] address three requirements but didn't provided a consistent addressing or caching scheme. Similarly, Microsoft's cache array routing protocol (CARP) [21] partitions the URL space among proxies. CARP uses hash-based routing to provide a deterministic request resolution path and eliminates the duplication of contents that otherwise occurs on an array of proxy servers. CARP made it possible to plug additional servers into the cache array or subtract a server from the array without massive reconfigurations and without significant cache reassigning. Its cache-management features provide both load balancing and fault tolerance. But it failed to deliver proportional tail content and missed on addressing one of our requirement.

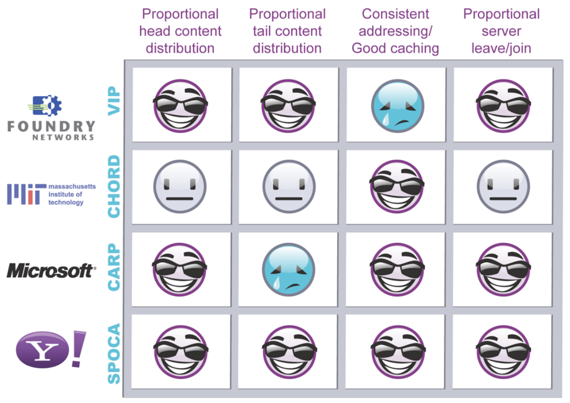
Proportional head content distribution
Proportional tail content distribution
Consistent addressing/ Good caching
Proportional server leave/join
FOUNDRY NETWORKS
VIP
massachusetts institute of technology
CHORD
Microsoft
CARP
Y!
SPOCA

Figure 16: Algorithm Comparisons

Our VIP routers do load balancing based on round robin servicing of connections. Network Dispatcher [11] was early work on this type of router. Round robin DNS is another way to do load balancing. These methods as well as other common methods are described in ONE-IP [5]. As we noted earlier these approaches balance the load, but do not use the aggregate memory of the cluster efficiently.

Caching services such as CoralCDN [8] and Akamai [15] use DNS resolvers to direct clients to caching proxies that are close to clients. Like SPOCA, they serve content to unmodified clients and are excellent at distributing popular content. Much of SPOCA's traffic is made up of a many requests for various unpopular con-

E: (Cache)	DAL	A2S	S1S
DLOD	8.2 days	4.5 days	40 days
WMOD	40 days	5.5 months	2 years
FLOD	9 months	1.4 years	1.5 years

Table 2: Statistics about churn times from the disk cache.

R: (RAM)	DAL	A2S	S1S
DLOD	2.5 hours	35 mins	40 mins
WMOD	1 hour	4.5 hours	3.8 hours
FLOD	4 hours	5.8 hours	6.4 hours

Table 3: Statistics about churn times from the memory cache.

tent that can pollute the front-end caches. For this reason SPOCA does not always choose to direct clients to local front-end servers for unpopular content.

The SPOCA router tries to use the aggregate memory of the front-end servers as one big cache. Locality-aware request distribution (LARD) [16] combined cooperative caching with request routing to achieve load balancing and effective cache utilization. LARD routes content based on load and uses a table indexed by the content identifier to consistently route requests. SPOCA's consistent routing function achieves load balancing without using a table entry for every cached object. We also handle popular content and failures with this same routing function.

As in other consistent addressing schemes [12, 23, 20], we assign each front-end server a section of an address space, and hash file names into this space in order to map files to servers. However, unlike any other scheme we are aware of, the address space is not completely assigned. Some addresses belong to no server. Specifically, a server is not responsible for all addresses between its own address and the address of whichever server is next in the hash space, as in distributed networks such as Chord [19]. Instead, each server is assigned a fixed section of the address space which is proportional to its capacity. Some systems [6, 1] which use consistent addressing mitigate the load balancing problems of Chord by mapping to many virtual servers using a hash function and then mapping sets of virtual servers to physical servers according to load. While this does allow coarse grained load balancing, it still does not handle popular content that needs to be served by multiple machines.

RUSH [10], scheme allows for cluster weights, thus insuring proportionality, but is not optimally consistent. When a cluster is removed or has its weight decrease, it not only causes the necessary shifting from itself to other clusters, it can cause shifting between other clusters. Also dynamic handling of hot streams is not covered.

Peer-to-peer networks are the most common applications of consistent addressing, but in peer-to-peer networks an appropriate partition of the address space is quite difficult to achieve. For example, in a 2005 technical paper, Giakkoupis and Hadzilacos [9] present a method of insuring that the largest section of the partitioned address space is no more than four times the size of the smallest section. Considering the complication that not all servers have equal capacity, their guarantee worsens to an eightfold imbalance between the most-loaded and least-loaded server, relative to each server's capacity.

Moreover, to achieve this factor-of-eight guarantee, Giakkoupis and Hadzilacos weaken the optimal consistency criterion, allowing up to twice the minimum content re-mapping when servers leave the cluster. This further underscores the tension between consistent addressing and proportional load balancing.

Consistent hashing [12] was proposed to handle popular content without swamping a single server. It extends work done by Harvest Cache [4] and Plaxton and Rajaraman [17]. The consistent hashing work was incorporated into a web cache [13] that used consistent hashing to route content requests to servers. Unlike SPOCA the web cache work does not use the same mechanism for load balancing, fault tolerance, and popularity handling. They also always serve requests close to clients, even if the content is unpopular.

8 Conclusion

Zebra and SPOCA routing simultaneously handles our requirements for load balancing, fault tolerance, elasticity, popularity, and geolocality. They do so using a simple mechanisms that nicely handle the different requirements in a consistent way.

Zebra and SPOCA do not have any hard state to maintain or per object meta-data. This allows our implementation not to have to worry about maintaining and recovering persistent state. It also eliminates any per object storage overhead or management, which simplifies operations.

Operations were also simplified by the ability to consolidate content serving into a single pool of servers that can handle files from a variety of different workloads. Further the ability to decouple the serving and caching of content from the storage of that content allowed us greater flexibility in architecting our caching and storage infrastructure. Specifically, this decoupling allowed us to have front-end clusters without any local content that only cache popular remote content.

In a for-profit corporation, cost savings and end user satisfaction are key success metrics. SPOCA excels on both accounts. We have seen great cost savings with a corresponding increase in the performance of our serving cluster.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, pages 1–16. USENIX Association, 2010.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] Brocade. http://www.brocade.com/index.page.

[4] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *USENIX*, 1996.

[5] O. P. Damani, P. E. Chung, Y. Huang, C. M. R. Kintala, and Y.-M. Wang. One-ip: Techniques for hosting a service on a cluster of machines. *Computer Networks*, 29(8-13):1019–1027, 1997.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazons highly available key-value store. In *SOSP '07: 21st ACM symposium on Operating systems principles*, New York, NY, USA, 2007. ACM Press.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8:281–293, June 2000.

[8] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *NSDI'04: 1st conference on Symposium on Networked Systems Design and Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[9] G. Giakkoupis and V. Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In M. K. Aguilera and J. Aspnes, editors, *PODC*, pages 302–311. ACM, 2005.

[10] R. J. Honicky and E. L. Miller. Rush: Balanced, decentralized distribution for replicated data in scalable storage clusters.

[11] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network dispatcher: A connection router for scalable internet services. *Computer Networks*, 30(1-7):347–357, 1998.

[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[13] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *WWW '99: Proceedings of the eighth international conference on World Wide Web*, pages 1203–1213, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[14] P. Linga. A churn-resistant peer-to-peer web caching system. In *ACM Workshop on Survivable and SelfRegenerative Systems*, pages 9–22, 2003.

[15] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, 2010.

[16] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. *SIGPLAN Not.*, 33(11):205–216, 1998.

[17] C. G. Plaxton and R. Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 570, Washington, DC, USA, 1996. IEEE Computer Society.

[18] M. Saxena, U. Sharan, and S. Fahmy. Analyzing video services in web 2.0: a global perspective. In *NOSSDAV '08: International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 39–44, New York, NY, USA, 2008. ACM.

[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 *SIGCOMM '01: conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[20] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw.*, 6(1):1–14, 1998.

[21] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.1. Internet draft, 1998.

[22] J. Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 1999.

[23] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In *SC '06: ACM/IEEE conference on Supercomputing*, page 122, New York, NY, USA, 2006. ACM.