NetCache Architecture and Deployment

by Peter Danzig, Network Appliance, Inc.

Abstract

This paper describes the architecture of Network Appliance's NetCache proxy cache. It discusses sizing proxy caches, contrasts the advantages and disadvantages of transparent caching, and reviews mechanisms to provide highly available caches. It also reports cache hit rates and summarizes our experience deploying proxy caching at Internet Service Providers (ISP) and corporate firewalls and intranets.

Contents

1. Introduction

When properly deployed, proxy caching reduces network bandwidth consumption and improves perceived network quality of service. However, when improperly deployed or inadequately sized, proxy caches degrade performance, need constant maintenance, and irritate users.

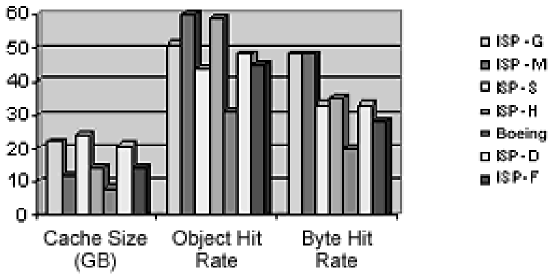

Chart 1: Hit rate of mission critical ISP & enterprise customers.

This paper summarizes four years of experience building and deploying proxy caches. NetCache springs from the Harvest cache project, which I led (the Harvest project still continues in the public domain under the moniker "Squid") [Harvest96]. What attracted us to web caches was our research that predicted that a hierarchical FTP cache would reduce Internet backbone traffic by 35%

[FTP93]. We proposed a hierarchical web caching protocol that became known as the Inter-Cache-Protocol [RFC-2186]. ICP lets cooperating caches robustly detect failure and recover from it quickly. As a team, we resolved to make the web scale more efficiently than the Domain Name System, which consumes 20 times more bandwidth than it really needs [DNS92]. We developed the Harvest cache in the public domain, and cooperative web caching swept through Europe, Asia, Latin America, and the Pacific.

To fill the need for a scalable, commercially supported, highly available web cache, we developed NetCache. The NetCache software versions run on UNIX and NT; the NetCache Appliance runs on Network Appliance's own Data ONTAP microkernel [TR-3014]. All versions of NetCache are built from a common source tree so their features are nearly identical. The high-end NetCache Appliance is roughly four times faster than the NetCache software running on a 2-processor, 300Mhz, Ultra-SPARC. The NetCache Appliance sits on top of Network Appliance's WAFL file system [Hitz94]. The appliance achieves superior performance because: (1) WAFL achieves 2-3 times more disk operations per second than traditional file systems, (2) because the ONTAP microkernel eliminates most of the data copying between the file system and network stack, and (3) because the appliance's event handling efficiently demultiplexes network connections. The NetCache Appliance survives disk failure transparently (without losing its configuration files, its log files, or even its cached contents) because it is RAID-based.

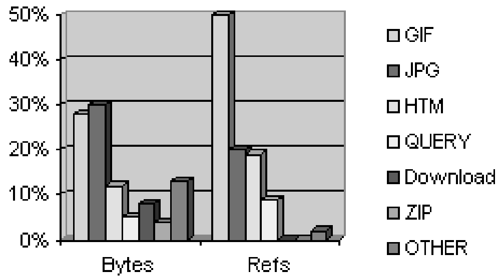

Chart 2: Why dynamic data minimally impacts hit rate.

We now have two years experience supporting NetCache for mission critical customers. As of December 1997, our biggest ISP customer had 500,000 dial-up customers and more than a hundred dial-in POPs. Our biggest enterprise customer had 100,000 desktop computer systems with browsers. Chart 1 shows cache sizes and hit rates of an assortment of NetCache ISP and enterprise customers. Cache sizes range from 6 to 28 GB and WAN bandwidths range from multiple T1 to T3. Bandwidth savings at these sites average 35%.

During the past two years, hit rates have remained stable, despite the proliferation of dynamic web content. The explanation for this is simple. Chart 2 shows that 70% of web traffic consists of graphic URLs and software downloads. Even if all "html" URLs were dynamic and non-cacheable, 80-90% of the web's bytes would remain cacheable. In the coming year, as HTTP-1.1 gets deployed, we anticipate higher hit rates as web servers begin exploiting cache-control headers.

After reviewing NetCache's architecture, we discuss how to scale NetCache to arbitrary WAN bandwidths and derive rules-of-thumb for sizing single nodes. Finally, we discuss the advantages and drawbacks of transparent caching.

2. Architecture

Figure 1 illustrates NetCache's architecture. Functionally, NetCache consists of: separate state machines to fetch WWW, FTP, and Gopher pages from their respective servers, state machines to tunnel HTTPS conversations, state machines to parse HTTP-1.1 requests [HTTP1.1], and state machines to map objects from and to disk. These state machines are driven by network, disk, and timeout events. Each state machine is uniquely bound to a single client, remote server, or disk file.

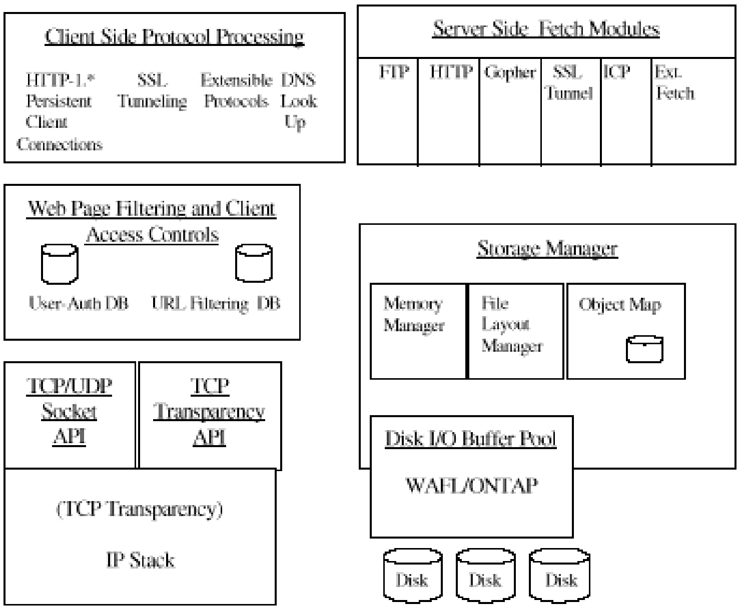

Figure 1: NetCache block diagram.

From a programmer's perspective, NetCache consists of sets of handlers for network, disk, and timeout events. These event handlers invoke routines in the storage manager, the server-side fetch modules, and the client-side protocol modules. Once called, an event handler runs to completion without the need for locking or multiprocessor synchronization. For coarse grain events, such as network and file system operations, the handler can dispatch work to an asynchronous thread and return. Communication between the main thread and the thread pool look indistinguishable from network or disk events.

2.1. Design Principles

We followed four principles to optimize NetCache's single node performance.

1) Avoid disk-ops: NetCache touches disk judiciously since every disk operation delays cache response time and traditional file systems usually only support about 50 disk-ops/s. NetCache maintains a small description of each cached URL in a memory resident, ½-MD5 indexed hash-table. The hash table lets NetCache determine whether or not a URL is a hit without touching disk. On reboot, NetCache re-constructs the hash table without touching file system meta-data. The table is sized so one cache node can support twenty-eight 4GB disks or fourteen 9GB drives. This amount of space is equivalent to 8 million 16-KB URLs.

The NetCache Appliance stores objects in Network Appliance's WAFL (Write-Anywhere-File-Layout) file system [Hitz94]. WAFL is a write-optimized, log-structured, RAID-aware file system with the unique property that its disk image is always consistent. This property allows the NetCache Appliance to reboot in a minute and to never need to check file system consistency (that is, no fsck). Data ONTAP supports hot-spare disks, so it is possible for the NetCache Appliance to rebuild failed disk drives without operator intervention. We optimized WAFL for web caching by exploiting the property that partially written URLs can be expelled upon reboot. We further optimized how it manages the meta-data of cached URLs.

The NetCache UNIX and NT software stores objects in files. We did not implement a NetCache specific file system atop raw disk partitions; instead, we wrote code to exploit the fast-path of the OS provided file system. On traditional file systems, creating and deleting files is ten times more expensive than reading and writing blocks. For this reason, as a rule, NetCache almost never deletes a file or disassociates the directory entry from the inode to which it's bound. NetCache's file allocation algorithm eliminates most of the costs associated with running on a traditional file system. The principal remaining costs are avoided by configuring the file system to not update file access time and to use asynchronous directory writes.

Sites that need the highest performance should run a NetCache Appliance. We could have avoided further disk-ops on our UNIX and NT versions if we had stored objects on raw disk. We did not do this. We believe that the maintenance burden of supporting our own file system would have violated our second design principle.

2) Keep it fast and simple: On all platforms, NetCache runs as a principal thread that dispatches work to a pool of slave threads. Profiling under high load shows that NetCache spends two thirds of its time in the TCP stack and I/O system, and less than a third in the "caching" application itself. For this reason, we threaded the disk and network I/O so that a single NetCache node can keep several processors busy executing TCP, disk I/O, and the NetCache application itself. Since it is more cost effective and fault tolerant to cluster many smaller caches than it is to deploy huge multiprocessors, we optimized NetCache on two-processor nodes.

To keep it maintainable, performance focused, and reasonably simple, we consciously chose not to make NetCache substitute for a firewall or an electronic commerce server. Instead, we created an extensible, performance oriented, architecture dedicated to caching web applications.

3) Scale by robustly partitioning workload: No single node web cache can scale arbitrarily; we profile and tune NetCache to run with up to 8,000 concurrent client, server, and file state machines. Beyond this, NetCache can cluster single cache nodes with its hashed-based cache-clustering algorithm, or with traditional ICP, to partition cache-to-cache workload and robustly recover from failure of parent caches. Partitioning workload with NetCache clusters decreases each individual cache's working set size and increases its hit rate. Partitioning workload with clusters takes maximum advantage of NetCache's persistent TCP connections and uniformly balances load across the members of the cluster. We discuss these issues in depth in Section 3.

4) Instrument it extensively: NetCache permits remote monitoring and configuration through password-protected, access-controlled web pages. The NetCache Appliance also exports an SNMP MIB. NetCache instruments and reports the cache object hit rate, the bandwidth saved by caching, and the URL response time as perceived by users. It reports the bandwidth delivered to the cache's clients and the bandwidth drawn from the WAN. It reports the health of NetCache itself by detailing the file system response time, disk I/O bandwidth, CPU usage, and memory consumed. NetCache reports the client request arrival rate, details of the cache replacement policy, the URL size distribution, and the most popular URLs. It also reports DNS response times.

NetCache writes its access log file in the "Squid" format, for compatibility with public domain log analysis tools. NetCache automatically rotates its access log. The access log identifies the URL, the client IP address or hostname, the access timestamp, and the HTTP server result code. Further, it also reports the time to look up the DNS record, the elapsed time to fetch the URL, and the number of bytes transferred. It indicates whether the URL referenced was a cache hit or miss, whether the

URL carried a cookie or was password protected and therefore uncachable. Finally, it indicates whether the client dropped the connection without fetching the entire URL.

2.2. Design Features

NetCache supports HTTP-1.1 persistent connections both to clients and to web and FTP servers. Persistent connections eliminate the need to setup and tear down a TCP connection per URL [Gettys97]. NetCache dynamically reduces the amount of time it will keep an idle persistent connection from 300 seconds down to to 1 second, as needed to constrain the number of open network connections. This wide dynamic range means that a lightly-loaded NetCache can keep its client TCP connections open for 5 minutes, yet a busy NetCache quickly relinquishes its idle connections so it does not run out of network connections. NetCache monitors and reports the effectiveness of its persistent connections. NetCache aborts any client or server connection that remains idle for more than 10 minutes so that jammed clients and web servers don't hog resources.

NetCache caches DNS records and actively ejects records that fail to work. NetCache uses the platform's default DNS resolver, but the NetCache API enables custom resolvers.

2.3. Filtering & Access Controls

NetCache supports a rich set of filtering, access control, and URL redirection primitives. If its native methods are not powerful enough, NetCache's API can express arbitrary access control policies as function of URL, MIME headers, and client identity.

Access Controls: NetCache supports IP-based access control lists for monitoring and administration, proxy access, and ICP access. NetCache also supports a separate administration port, bound to "localhost," that enables encrypted, remote management via "ssh" tunnels. Access controls can be tailored to specific clients by mapping client identity into the reverse DNS lookup.

Filtering: NetCache blocks and redirects URLs via a hashing algorithm that introduces negligible overhead through several million blocked sites. NetCache fast-filtering hashes a URL's hostname or IP address into a hash bucket and then walks the chain to determine whether or not to block the particular URL requested. You can exclude both subtrees (http://www.netapp.com/priv/) and individual (http://www.netapp.com/netapp.gif) URLs. Hundreds of rules can be specified for a particular hostname at negligible cost.

NetCache supports regular expression-based filtering. In contrast to fast filtering, filtering with regular expressions impacts performance when given thousands of expressions. NetCache's API lets you plug in commercial blocking software such as Surfwatch or write your own access control scheme.

2.4. Consistency

NetCache guarantees that it returns up-to-date URLs via HTTP-1.0's GET if-modified-since operator. NetCache will eventually support HTTP-1.1's opaque cache validators. NetCache can be configured to verify on every reference or, for faster responsiveness, to verify the URL only once every few hours. This has the potential to serve stale data at the benefit of reducing response time from seconds to milliseconds. NetCache will not cache URLs whose MIME headers specify negative expire times, that carry "cookies," that appear to be the result of queries or other "cgi" programs, or of password protected pages. For end-to-end consistency, if the browser sends a reload, NetCache flushes and re-fetches the URL

NetCache does not validate pages on its own accord because this causes work without guaranteeing consistency. NetCache does not speculatively pre-fetch URLs before they are referenced because log analysis shows that this speculative pre-fetch does not make up for the bandwidth and cache resources it consumes.

2.5. Lights-Out Operation

The NetCache Appliance is designed for lights-out operation. It supports redundant power supplies, hot-spare disks, and access-controlled remote administration. Software upgrades on the NetCache Appliance can be done with a web browser (no floppies, CDs, or flash cards). If the upgrade should fail to boot, the appliance's PROM monitor automatically fails back to a saved version. A serial port enables emergency remote administration from a terminal server. In addition to SNMP, the appliance can automatically send email when disks need servicing or when hardware or software errors are encountered.

3. Scalable Caching

You can make a web cache scale by partitioning browser workload and by partitioning and aggregating cache-to-cache workload hierarchically.

3.1. Partitioning Browser Traffic

Proxy auto-configuration is the most cost-effective and easiest way to partition browser workload, and both Netscape and Microsoft browsers support it. Proxy auto-configuration transparently masks cache server failures, provides a mechanism to bypass caching during server maintenance, and is a convenient place for specifying workarounds for non-compliant web servers that interact badly with the caches. Auto-configuration doesn't offer fine-grained load balancing, but this is usually unnecessary with properly sized individual cache nodes

For finer load balancing and to wrestle control over failover from the browser, you can combine proxy auto-configuration with a load balancing TCP router (Network World recently contrasted four such solutions [NW97]). Combining the two lets you partition workload and retain ownership over how the web caches failover.

A load balancing TCP router follows three steps to map a client's TCP session to a ready web cache. First, it responds to the client's TCP SYN packet on behalf of the web caches and establishes a TCP session with the client. Second, it selects a web cache and creates a new TCP session with it. If it cannot create the TCP session with this cache, it tries to connect to alternative caches until succeeding. Third, once both TCP sessions are established, it routes the client's TCP packets to the cache by translating sequence and port numbers such that the two endpoints believe that they are communicating directly with one another.

We recommend an auto-configuration function that partitions client traffic by the URL's hostname. This best exploits NetCache's persistent TCP connections. If the partition function depends on more of the URL, subsequent URLs within given web servers will map to different caches and therefore defeat the advantages of persistent TCP connections.

Consider the `proxy.pac` function below [PAC96]. It uses a simple hash function to route URLs to four caches. Each cache is responsible for the particular URLs that map to the hash bucket. Since caches don’t replicate URLs, disk requirements and network consistency traffic between caches and web servers is reduced. Note that in contrast to OS- provided "clustering" solutions, the individual cache nodes need not communicate with one another, eliminating significant overhead.

As a convenient hash, `proxy.pac` uses the string length of the URL’s hostname. It specifies a primary and a secondary cache for each "hash" bucket. Should both the primary and secondary cache be unavailable, the browser resolves the URL directly. If the hash function is not restricted to the hostname, performance drops because subsequent URLs within a page may be sent to different caches. Each cache involved needs to establish its own persistent TCP connection to the requested web server, reducing performance and adding latency. Our experience shows that this hash is well balanced.

```
function FindProxyForURL(url, host)
{
        var hash = 0;

        hash = (host.length %4);

        if (url.substring(0,5) == "https")
          return "DIRECT"

        if (url.substring(0,5) == "snews")
          return "DIRECT"

        if (hash == 0)
          return "PROXY 192.168.1.1:8080;
                                 PROXY 192.168.1.2:8080;
                                 DIRECT"

        if (hash == 1)
          return "PROXY 192.168.1.2:8080;
                                 PROXY 192.168.1.3:8080;
                                 DIRECT"

        if (hash == 2)
          return "PROXY 192.168.1.3:8080;
                                 PROXY 192.168.1.4:8080;
                                 DIRECT"
        else
          return "PROXY 192.168.1.4:8080;
                                 PROXY 192.168.1.1:8080;
                                 DIRECT"
}
```

Notice that browser proxy auto-configuration fails over without the need for router or clustering support. Additionally, the NetCache Appliance will support two-node clustering in the future, and clustering software is already available for the NetCache software [Solos97].

Note that the next generation of load balancing routers will parse the URL before passing the TCP connection to the web caches [ArrowPoint]. When this technology becomes available, the router, rather than the auto-configuration script, can partition client requests.

At a single node cache, auto-configuration failover can specify that clients fetch URLs directly or, perhaps, fetch them from an upstream cache. Given dozens of cache nodes, load balancing routers and auto-configuration can be combined to both partition client traffic and load balance and failover cache servers.

3.2. Partitioning Hierarchically

NetCache partitions cache-to-cache workload so that hierarchical caching can scale, using a hashing scheme similar to the `proxy.pac` in Section 3.1. NetCache supports traditional ICP, but also optimizes it as described below.

3.2.1. NetCache Clusters

NetCache clusters uniformly partition cache-to-cache traffic so that peer caches are responsible for non-overlapping URLs. Figure 2 illustrates the two ways to use NetCache clusters.

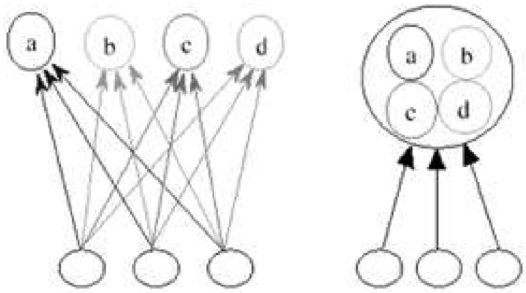

Figure 2: Clustering hierarchically (left) and cluster caching (right).

On the left side of Figure 2, we see a set of child caches connecting to a set of parent caches. The child caches declare their parents as a cluster cache. This causes the children to partition their requests uniformly and map each URL onto a unique parent cache.

NetCache retains persistent TCP connections between child and parent caches. In lieu of using ICP to detect parent failure, a NetCache child detects parent failure when it fails to initiate a persistent connection with it. If it fails to establish a TCP session within two seconds, the child declares the parent dead and immediately fails over to an alternate parent if one exists or, if not, fetches URLs directly. After five minutes, instigated by a new request, the child probes the failed parent. It declares the parent recovered if it establishes a TCP connection with the parent within two seconds.

The advantage of NetCache clusters over traditional or multicast ICP is that it avoids ICP's network round trip time. It evenly partitions workload among the cache parents, yields high hit rates, and minimizes cache consistency traffic. Eliminating the ICP round trip time saves anywhere from 100 to 2000 ms per URL fetched. Its uniform partitioning eliminates the chore of manually partitioning the parent caches by domain name. Hit rates are higher and consistency traffic is lower since every URL maps to a unique cache.

The right side of Figure 2 illustrates the second way to use NetCache clusters -- a substitute for browser-based partitioning when it is not available. In this method, clients connect to any NetCache through a load-balancing router or with simple DNS rotation. Each NetCache, in turn, fetches cacheable URLs through the cluster member to which the URL hashes. The hash follows an algorithm similar to the `proxy.pac` previously discussed. Cluster caches use the same replacement policy for URLs that map to themselves as they do for URLs that map to other caches. For this reason, frequently referenced objects get cached on disk of many or all nodes of the cluster, but less frequently referenced objects only stick to the cache to which they map. The cost of cluster caching is node-to-node communication that proxy auto-configuration otherwise avoids.

Our experience shows that deploying NetCache clustering yields nearly the same hit rate as browser-based partitioning. However, since browser-based partitioning eliminates data transfers between cluster-members, it is more efficient than server-based clustering. If you can partition the browser traffic, then you do not need to enable this form of clustering.

3.2.2. Low Bandwidth and ICP

Traditional ICP lets you construct robust cache hierarchies and share cache objects between loosely or tightly coupled caches. ICP is used primarily in limited bandwidth scenarios. We illustrate each of these properties by example.

Robustness: One of our ISP customers purchases bandwidth from three service providers (say MCI, Sprint, and UUNET) so that they can tolerate failure of any two providers without interrupting service. Before deploying ICP-enabled web caches, they load-balanced these links by assigning client network numbers to three groups and configured their BGP routing protocol software to advertise different weights for each of the three groups. The provider for the first address group defaults to MCI, the second to Sprint, and the third to UUNET. When BGP detected that the link to MCI failed, it adjusted routing so the first group switched to UUNET or Sprint.

Although their solution was simple and functional, it didn’t respond quickly to transient, but frequent link failure and their customers complained of intermittent errors when browsing the web. Their solution failed to aggregate the combined bandwidth of their three service providers, and they had to load balance their individual links by shifting the membership of the address groups.

When they switched to web caching with ICP, they benefited from instant recovery from link failures and automatic, dynamic load balancing of their links. They configured their web caches hierarchically with upstream caches at their respective service providers. ICP let them simultaneously harden their connectivity and increase aggregate throughput, adding robustness.

Cache Sharing: Another of NetApp’s ISP customers operates dozens of physical points-of-presence (POPs) and routes traffic through a handful of data centers (or "hubs"). They place primary caches at the POPs and secondary caches at the data centers, as illustrated in Figure 3. Between their POPs and data center caches, they configured NetCache’s hierarchical clustering to exploit NetCache’s persistent connections and load balancing. At their hub, their caches respond to traditional ICP requests so that they can sell "hits" to their smaller competitors who peer with them by ICP. Their competitors fetch their own parent cache misses but pull hits from our customer. ICP lets them distinguish hits from misses inexpensively.

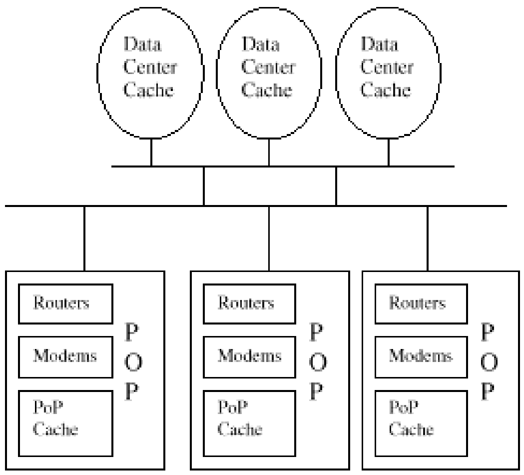

Figure 3: Hierarchical, clustered caching between a POP and a hub or data center.

You can mix NetCache clusters and traditional ICP neighbors and parents, although this forces the cluster to communicate via ICP, which degrades performance and does not fully exploit the persistent connections between caches.

3.2.3. URL Routing

In addition to clustering and ICP, you can specify URL patterns that must be fetched directly or through particular caches or clusters. For example, NetCache can route internal URLs differently from requests for external URLs. You can specify a set of URLs that must be resolved directly, avoiding up-stream caches. You can force certain URLs to be routed through particular up-stream caches to traverse firewalls or to force the URL through a particular transcoder.

4. Sizing Individual Caches

Sizing a web cache's memory, disk, and CPU resources depends on its workload and WAN bandwidth. Below, we derive rules-of-thumb for scaling, based on conservative assumptions. Deployments scaled to these rules should be stable through one doubling of the applied workload.

Disk: Let's assume that the average size of a cacheable URL is 8 KB. (The real median is a bit smaller, and the real average is usually somewhat higher than this). We use 8 KB to be conservative. A 1-Mbit/s link can carry fifteen 8-KB URLs per second. Assuming a 4-KB disk block size, each URL averages two disk operations to write or read the data and, at most, one disk operation to access the file system meta-data. In general, we do not want to run the disks above 50% utilization, to keep disk queue lengths insignificant.

Planning for 3 disk-ops per URL and 15 URLs per second yields 45 disk operations per second per megabit of WAN bandwidth. Given that a SCSI disk can sustain 50 ops/s, planning for 50% utilization requires two SCSI disks per megabit of WAN bandwidth on UNIX NetCache. The NetCache Appliance obtains 120 disk-ops/s, so it needs only one disk per Mbit/s. In the event that disk is under-provisioned, both the NetCache Appliance and the software versions do not present a bottleneck: NetCache bypasses disk if the disks fail to keep pace with the network.

The amount of disk storage depends on link speed, too. Typically, you want enough disk space to store a week of WAN traffic. If you reference 1 GB of traffic per day at a 25% bandwidth savings, then you want a 5-GB cache (75% of 7 GB). In our experience, average daily link utilization is frequently a quarter of peak utilization. This means that a customer with a 1 Mbit/s connection references 2 GB of data per day or 14 GB per week. Deploying two 4-GB or 9-GB disks should be sufficient.

Deploy two disks per Mbit/s of WAN traffic. On the Appliance, deploy one disk per Mbit/s.

Memory: NetCache uses 10-40 bytes of real memory per cached URL for its in-memory meta-data. Assuming 8-KB URLs, NetCache stores 1 million URLs per 9-GB disk and consumes 10-40 MB of meta-data. Each concurrent client consumes two TCP buffers and each server connection consumes two more. Assume 8-KB TCP buffers. NetCache itself consumes 64 KB of memory per client connection. Finally, averaging over both fast and slow connections, assume that the time to fetch a URL averages 10 seconds. This may seem high, but the average of four 1-second requests and one 46-second request is 10 seconds; the average response time grows quickly.

Little's Law states that the number of requests in service, N, equals the product of the request arrival rate, l, and the total time, W, that a request remains in service [Kleinrock75].

N=lW

At 15 URLs per second, Little's Law says that that for every Mbit/s of WAN traffic, there are 150 concurrent client connections. Hence, for every Mbit/s of WAN traffic, we need 5 MB for TCP buffers, 10 MB for client data structures, and 10-40 MB for NetCache data structures.

Deploy 50 MB of RAM for each Mbit/s of WAN traffic.

Processor: NetCache on UNIX supports 4,096 connections. The NetCache Appliance supports 8,192 connections. Because each concurrent request consumes a client and a server (or file) connection, each UNIX NetCache supports 2,000 different client connections and each NetCache Appliance supports 4,000. A 2 processor, 300 Mhz Sun Ultra-2 has enough CPU to drive about 5-8 megabits of WAN while serving 2,000 concurrent client connections. The NetCache Appliance can drive 15-20 Mbit/s of WAN traffic while supporting 2,000 client connections. Note that the NetCache Appliance can saturate a 100-Mbit/s network link, in the lab. However, we recommend a more conservative rule when deploying it.

Deploy one NetCache software unit for every 5 Mbit/s of WAN traffic.
Deploy one NetCache Appliance for every 15-20 Mbit/s of WAN traffic.

Ports: When the WAN suffers congestion, a cache that handles 150 concurrent clients under good conditions can find itself handling 500 connections or more as more and more browsers become active. This leads to the final rule.

Deploy one NetCache software unit for every 1,000 ports.
Deploy a NetCache Appliance for every 2,000 ports.

5. Deploying Caching

After an ISP deploys proxy caching, how does it train its users to configure caching? The most flexible way is to deploy new browser releases that are pre-configured with a proxy auto-configuration URL. After two or three browser releases, the majority of frequent users will be cache-ready. After caching is deployed, edit `proxy.pac` to partition URLs across the caches. Proxy auto-configuration maps URLs to specific proxy servers and identifies which URLs should be fetched directly.

Even ISPs with multiple POPs need only configure a single auto-configuration URL into the browsers. One way to do this is to program a web server to compute an appropriate auto-configuration script based on client IP address. Another way is to reserve a common block of private IP addresses and use these same addresses at each POP.

Another alternative is to deploy transparent caching.

5.1. Transparent Caching

A transparent cache acts much like a gateway or firewall -- it effectively sits between the users and the network. The advantage of transparent caching is that it eliminates the need to configure browsers to use caching. Another strength (and sometimes a weakness) is that it is impossible to bypass caching.

Conceptually, transparency works by modifying the TCP/IP stack of a cache so that it operates in "promiscuous mode" and effectively binds itself to all possible IP addresses. In the NetCache appliance, we literally modified the cache’s TCP/IP stack to be promiscuous. In the NetCache software, we employ network-address translation to make the TCP/IP stack promiscuous.

5.2. Strengths

The strength of a transparent cache is that clients cannot easily bypass it. Regardless of whether or not a sophisticated user disables proxy auto-configuration, a transparent cache intercepts the user’s requests. The great strength of transparent caching is the ease in which it can be deployed and made scalable and tolerant to cache failure.

NetCache can be deployed with an L4 switch to make a scalable and fault tolerant transparent cache. The L4-switch partitions the cache workload into non-overlapping buckets. This guarantees a good hit rate since the same URL hashes into the same bucket reference after reference, in a manor analagous to the `FindProxyForURL()` script described earlier. When a cache fails, the L4 switch either sends its traffic directly through to the WAN or diverts it to another cache.

NetCache can be transparently deployed directly with a Cisco router to create a scalable and fault tolerant transparent web cache. NetCache communicates its state to the router so that the router can cut requests directly to the net if a cache fails.

We discuss the details of transparent caching in [TR-3033].

5.3. Weaknesses

Transparent caches are a bit slower than non- transparent ones because standard caches perform and cache their DNS lookups, and because browsers maintain persistent connections to traditional caches but drop their persistent connections to transparent caches as soon as the browser browses a different web site. Transparency can add 200 milliseconds to the response time of modem speed clients.

Potentially decreased network robustness is transparency’s greatest weakness. Transparency breaks the Internet’s single founding principle that network connectivity is end-to-end. Transparent caching depends on the absolute stability and symmetry of network routing. If routing through a transparent cache becomes asymmetric, the network can appear disconnected for HTTP traffic (yet connected for other protocols). In contrast to Internet design, when routing flaps, users of transparent cache may see broken web pages as their HTTP conversations are no longer end-to-end.

Most transparent caches do not cache FTP. Some 10% of cacheable network bandwidth is due to FTP.

Because transparent caches are difficult to bypass, it’s also hard to deploy quick fixes when a particular web server interacts badly with the cache software. With proxy auto-configuration, it’s easy to bypass caching for particular web servers. There’s also a client-side work around when things go wrong: disable caching. With transparent caches, workarounds may require upgrading cache software or propagating new policy-based router configuration rules, which requires great care and can degrade router performance.

We use transparent NetCache in Network Appliance's corporate intranet. Many of our ISP customers use transparent caching at their POPs and data centers. We recommend that transparency be deployed only in locations where network routing can not flap between alternative routes.

6. Summary

NetCache software serves 100-200 URLs/second on the same hardware that public domain servers such as Squid can serve 25-50 URLs/second. The NetCache Appliance, because of its close integration with the file system and Data ONTAP microkernel, achieves four times the performance of the NetCache software, peaking at about 500 URLs/second.

References

[ArrowPoint]
Product announcement dated November 10, 1997.

[DNS92]
Peter Danzig, Katia Obraczka, and Anant Kumar, "An Analysis of Wide-Area Name Server Traffic: A study of the Domain Name System," *1992 ACM SIGCOMM Proceedings*, pp. 281-292.

[FTP93]
Peter Danzig, Mike Schwartz, and Richard Hall, "A Case for Caching File Objects Inside Internetworks," *1993 ACM SIGCOMM*.

[Gettys97]
Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud´hommeaux, Håkon Lie, and Chris Lilley, "Network Performance Effects of HTTP/1.1, CSS1, and PNG," *Proceedings of the 1997 ACM SIGCOMM*.

[Harvest96]
Anawat Chankhunthod, Peter Danzig, et. al., "A Hierarchical Internet Object Cache," *Proceedings of the 1996 USENIX Annual Technical Conference*, pp. 153-163, San Diego, January 1996.

[Hitz1994]
Dave Hitz, James Lau, and Michael Malcolm, "File System Design for a File Server Appliance," *Proceedings of the 1994 Winter USENIX Technical Conference*, pp. 235-245, San Francisco, January 1994. Also published by Network Appliance as TR-3002.

[HTTP1.1]
R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, T. Berners-Lee, *Hypertext Transfer Protocol - HTTP/1.1*, November 21, 1997.

[Kleinrock75]
Leonard Kleinrock. Queuing Systems, Volume 1, Wiley, 1975.

[NW97]
Rich Farrell, "Review: Distributing the Web Load," *Network World*, September 22, 1997. This article reviews load-balancing routers from Cisco, F5 Labs, RND Networks, and HydraWeb Technologies.

[PAC96]
Proxy auto-configuration, Netscape.

[RFC-2186]
Kim Claffy and Duane Wessels, *Internet Cache Protocol (ICP), version 2 (RFC 2186)*, September 1997.

[RFC-2227]
J. Mogul and P Leach, *Simple Hit-Metering and Usage-Limiting for HTTP (RFC 2227)*,

October 1997.
[Solos97]
Solos Arthachinda, Katia Obraczka, and Peter Danzig, http://www-scf.usc.edu/~arthachi/clusterd-doc Software fail-over (clustering) for web servers and caches.
[TR-3014]
Andy Watson, *Multiprotocol Data Access: NFS, CIFS, and HTTP (TR-3014)*, Network Appliance, Mountain View, California, 1996.
[TR-3033]
Peter Danzig and Karl Swartz, *Transparent, Scalable, Fail-Safe Web Caching (TR-3033)*, Network Appliance, Santa Clara, California, 1998.
