Adaptive Web Caching

Principle Investigators:
Lixia Zhang (UCLA),
Sally Floyd and Van Jacobson (LBNL)
February, 1997.

Technical rationale, technical approach, and constructive plan

The success of the World Wide Web has brought an exponential growth of the user population, the total host count, and the amount of total traffic volume on the Internet. As the Internet connectivity is reaching the global community, the World Wide Web is becoming a global-scale data dissemination system. Inevitably, this over-night exponential growth has also caused traffic overload at various places in the network. Until recently, advances in delivery fabrics gave the impression that scaling the Internet was simply an issue of adding more resources. Bandwidth and processing power could be brought to where they were needed. The Internet's exponential growth, however, exposed this impression as a myth. Information access has not been, nor will it likely be, evenly distributed. As have been repeated observed, popular Web pages create "hot spots" of network load, with the same data transmitted over the same network links again and again to thousands of different users. Hot-spots also move around. The photographs of Venus congested Los Nettos for one week; the "Midnight Madness" release of Microsoft's Internet Explorer 3.0 congested NorthWestNet for 48 hours, threatening Internet connectivity to University of Washington These are but a couple of well publicized examples. Recent studies by Margo Seltzer of Harvard University also confirms that flash-crowds are very common, and that the "cool site of the day" moves around [6]. Bottleneck hot spots develop and break up more quickly than the network or the Web servers can be re-provisioned. A brute force approach to provisioning is not only infeasible, but also ineffective.

The lessons of twenty-five years of Internet experience teach us that caching is the only way to handle the exponential growth of user demands. Seltzer's study also shows that the more popular the pages, the less likely they are to change. Similarly, the larger documents are less likely to change than smaller ones. Instead of always fetching pages from the originating source, data requests can often be more effectively answered by finding "local" copies near consumers.

We need to develop a new infrastructure for data dissemination on an ever-increasing scale. We believe that a multicast-based adaptive caching infrastructure can meet this challenging need. In the rest of this section, we first outline an ideal adaptive caching system. We next describe a multicast-based design to realize the desired functionalities. We discuss in detail the two main issues in building the proposed system, autoconfiguration of cache groups and automatic forwarding of Web requests through this maze of cache groups.

A Dream Picture

Given that the basic problem is data dissemination to thousands or millions of users, the basic solution ought to be some form of multicast delivery. That is, when multiple users are interested in the same data, the data should be fetched only once from the origin server, and then forwarded via a multicast tree to all the interested parties. Ideally, each piece of data would travel through each network link no more than once.

Unlike multicast delivery for realtime multimedia applications, however, Web requests for the same data come *asynchronously* because different users surf the Web at different times. Therefore Web "multicasting" must be done via caching: the network temporarily buffers popular Web pages at places the pages have traveled through (due to previous requests), so that future requests for those pages can be served from the cache. Benefits of caching include reduced load at origin servers, shortened page fetching delays to end clients, and best of all, reduced network load which reduces potential congestion.

One big challenge in building such a caching system is that, generally speaking, we do not know beforehand which pages would be interesting to users, or where the interested parties may be located, or when they may fetch the pages. Following the basic principles in the Internet architecture design, we propose to build an *adaptive* caching system. Ideally, we envision a caching system in which a popular page would automatically walk itself down its distribution tree in response to the intensity of requests. The higher the demand for the page, the closer the page would get cached to end users and the more copies made; furthermore, the fetch requests for that page would automatically discover the nearest cache copy. On the other hand, pages that are rarely fetched would not leave their origin servers, or walk very far down the distribution tree.

Another challenge in building this caching system is that a popular Web site may pop up anywhere at any time in the Internet, a number of popular sites may all exist at the same time, and different data is hot at different places[1]. If the distribution paths for each page make a tree, and multiple trees exist simultaneously, each rooted at the origin server of a popular page, clearly the caching infrastructure to be built cannot be a tree itself. Instead, the infrastructure ought to be a mesh on which cache trees can automatically build themselves as popular pages are pulled down towards their clients. As time goes, the trees should also automatically vanish as the pages become a past interest.

The Basic Approach

The previous section may have painted a seemingly impossible system to build. In this section we describe how IP multicast can be used as the basic building block that enables us to realize this dream system. The example topology in Figure 1 is used to illustrate our basic design in this section.

[1] From Seltzer measurement.

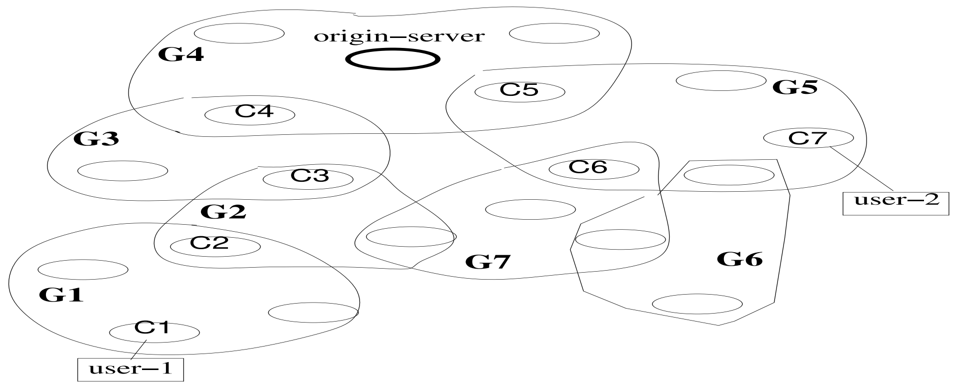

Figure 1: An illustrative example of our adaptive caching design.

Use Multicast

IP multicast serves two distinguished functions, one being the most efficient way to deliver the same data to multiple receivers, the other being an information discovery vehicle—a host can multicast a query to a relevant group when it does not know exactly whom to ask. Our caching design makes use of both features; we multicast page requests in order to locate the nearest cache copy, and multicast page responses in order to efficiently disseminate pages that have common interest.

To find the nearest cache that holds a requested page, the simplest approach could be to have all the Web servers and cache servers join a single multicast group. Then one could simply multicast a page request to that group. The nearest cache or origin server with the page will be the first one to hear the request and respond. One fatal flaw of this simple approach, however, is that it does not scale—we simply cannot afford multicasting all Web requests globally.

One scalable version of the above idea is to organize all Web servers and cache servers into *multiple* local multicast groups[2], as shown in Figure 1. When `user-1` requests a new page, it sends the request to a nearby proxy C1, which is also a cache server. If C1 does not find the requested page in its local cache, it multicasts the request to a nearby local group of which it is a member; in the example the nearby group is G1. It is possible that some cache in G1, say C2, has the requested page in its local cache, in which case C2 multicasts the requested page to G1, and C1 will forward a copy back to `user-1`. However in case of a cache miss within the local group, the request must be further forwarded, as explained next.

Request Forwarding

To handle the request forwarding problem, we propose that cache servers join more than one multicast group, so that all the cache groups heavily overlap each other. When there is

[2]Although the groups are made of both Web servers and cache servers, in the rest of this paper we call them cache groups.

a cache miss in one group (as indicated by the lack of a response message), each cache of the current group checks to see if its other group(s) lies in the direction towards the origin server of the requested Web page. In our example of Figure 1, C2 would realize that its other group, G2, lies in the direction to the origin server. When a cache finds itself in the right position to forward the request, it also informs the current group when doing so. This forwarding decision may also take into account such factors as the past history of neighboring cache groups in answering previous requests. We must also handle cases when no cache in the group volunteers to forward the request.

In case the second cache group has a miss again, the request will be forwarded further following the same rules (in Figure 1, for example, C3 will multicast the request to G3). Proceeding in this fashion, the request either reaches a cache group with the page, or otherwise is forwarded through a chain of overlapping cache groups between the client and the origin server, until it reaches the group that includes the origin server of the requested page.

Page Retrieval

Once the request reaches a group in which one or more servers have the requested page, the node holding the page multicasts the response to the group, possibly after a short random wait using an algorithm similar to the one developed in Scalable Reliable Multicast (SRM) [3]. This multicast response loads neighboring caches in the same group with the page. For example, when a short time later another request for the same page comes from `user-2` to group G5, the request will stop one "multicast hop" short of the group with the origin server (G4). The cache that is the member of both G4 and G5, namely C5 in this case, can now multicast the page to G5 (see Figure 1).

The original request gets fulfilled as follows: when cache C4 gets the response, it will relay it back via unicast to the node from whom C4 first heard the request, which in this case is C3. In this way the response page will be relayed back to the original client by traversing those cache servers that multicast-forwarded the request earlier. To further speed up the page delivery, an alternative is to let C4 open a HTTP connection directly back to `user-1`'s proxy server C1.

A couple of issues deserve further discussion here. First, although the response is multicast to the local group whenever a hit occurs, we assume that each cache in the group decides independently which pages to save. Multicasting the response to the local cache group can be done reliably using SRM. SRM supports receiver-driven reliable delivery, thus it provides flexible support for selective reliability. Caches in the local group that are interested in reliably caching the data will request retransmission for any corrupted or lost data; uninterested parties simply ignore all this.

Another issue concerns data integrity. Hop-by-hop page forwarding through a chain of unknown caches increases the risk that the data may be intentionally or unintentionally corrupted. Such potential danger, however, is not new due to caching. In today's Internet, hop-by-hop packet forwarding through unknown intermediate routers could also impact data integrity inadvertently; our proposed caching design simply mimics the "store-and-forward" packet delivery at a higher level. However, it is true that the addition of caching introduces new opportunities for things to go wrong. We believe that the fundamental solution to the data integrity problem is end-to-end integrity checking via mechanisms such as MD5 checksum.

Seeing a Demand-Driven Data Diffusion Yet?

From the above we see that fetching a Web page the first time from the origin server has a nice side-effect: the requested page is multicast to the group wherever there is a "hit" to achieve the effect of "popular Web pages walking themselves down the cache tree". In this fashion the servers in the same group with the origin server of the page are loaded with that page. If subsequent requests for the same page come in within a short time period (before the cached object expires or gets deleted), they will see a hit before reaching the group with the origin server. Each of these hits causes the page to propagate "one hop" away from the source and get closer to end clients. Thus popular pages quickly propagate themselves into more caches in the distribution trees. Pages with infrequent requests, on the other hand, will be seen only by a few caches near the origin server.

We expect further engineering tuning of the design parameters once we get the first implementation up and running. For example, if the deployment starts with a limited number of cache groups, then one may want to multicast the page to the local group only after seeing consecutive requests for the same page within a short time interval. If the world eventually ends up with a large number of cache groups, then pages moving one multicast-hop away for each hit may be exactly the right speed. The speed of data diffusion through caching represents an engineering tradeoff among various factors, but the goal remains the same: an adaptive system that loads itself according to the demand.

Hierarchy and Scalability

Generally speaking, a scalable system requires some sort of a hierarchical structure. What we proposed above, however, is a mesh of overlapping groups, rather than a strict hierarchy. It is on the base of this overlapping mesh that each popular page grows its own cache tree. Cache servers themselves, on the other hand, do not know or care about the contents of the pages they cache, or how many distribution trees they have been on. They cache pages strictly based on the popularity of the demands, a property that enables our design to scale well with large user populations. Our design is in sharp contrast to some other proposed cache schemes where the performance relies on analyzing *individual users'* page fetching patterns and pre-loading pages accordingly.

The cache tree for each popular page may come and go highly dynamically, but the cache groups remain relatively stable. As described in the next section, cache groups adjust themselves over time according to observed changes in topology, workload, and user population. When user population and page demand grow, the number of caches and/or the caching power will need to grow accordingly. Our design will let this cache infrastructure automatically readjust itself to meet the new load demand.

Autoconfiguration of Cache Groups

In order for this infrastructure of Web caches to be both scalable and robust, the organization of Web caches and servers into overlapping multicast groups must be self-configuring, for several reasons.

- Manual configuration does not scale, as evidenced in the SQUID system.
- Manual configuration tend to be error-prune.
- Self-configuring capability enables cache groups to dynamically adjust themselves according to changing conditions in network topology, traffic load, and user demands, thus achieving the goal of both robustness and efficiency.

We believe that self-configuring systems are an essential component for a range of large-scale systems in the Internet. Examples include the need for self-configuring groups for session messages in RTP, the need for self-configuring groups for session messages and for local recovery in scalable reliable multicast (SRM), and the need for self-configuring search structures for information discovery protocols. We envision that the basic approaches to self-configuration developed in our Web caching design can be further extended to other loosely-coupled, large-scale information dissemination systems.

We envision a world in which clusters of caches are placed at both network access points and internally throughout the various autonomous networks in the Internet. Through a cache group management protocol (CGMP) to be designed, all Web servers and cache servers automatically organize themselves into geographically and administratively *overlapping* groups. Because one basic function of a cache is to relay requests and responses between groups, it is highly desirable that cache servers run on multi-homed hosts. They can then easily join different multicast groups, one on each of their network interfaces.

A critical task in building the proposed adaptive caching system is to design this Cache Group Management Protocol (CGMP). The autoconfiguration of cache groups must satisfy a number of contrasting constraints. On average, a request for a new page only needs to travel a small number of "hops" along some "shortest path" to reach the origin server. Thus, the cache groups must have both adequate size and adequate overlap among the groups. On the other hand, as a cache group becomes larger in size, the group's traffic, overhead, and workload increase. The configuration protocol needs to balance these requirements for a small number of cache resolution hop counts and low overhead within each cache group. The cache groups must also be able to dynamically adjust to the addition or deletion of caches, routing and load changes, application performance, and tolerance to overhead.

The basic functionality for cache group managements concerns group creation and maintenance. This includes regrouping according to the observed group load, the group cache hit ratio, the tolerance of group overhead, and the change in topology and caches. For example, a cache group could split when there is too much traffic in the group, or a cache group with a low hit ratio could merge with another group.

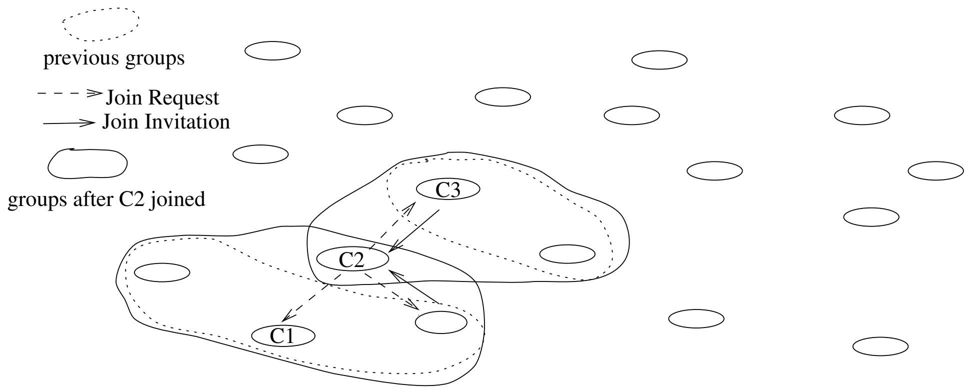

Figure 2: An example of group management.

Group Creation

We plan to start with the group management protocol developed by MASH project [12] as the base and gradually evolve that protocol to our cache group management protocol. The basic idea has the following steps:

- A well-known multicast address (WKM) is assigned for cache group usage.
- When a new cache, C2, starts up, it performs a expanding ring search for existing Web groups around its neighborhood by multicasting a *Group Join* request to WKM out each of its network interfaces. The request may be repeated with an increasing TTL value until some neighboring groups are found, or until C2 gives up (see later).
- When an existing cache, C3, hears this request, C3 sends a reply with its own group address as an invocation to C2 to join the group. This assumes that C3's group G2 is not overly full (as described later).
- C2 joins the cache group from which it receives an invitation. If C2 receives more than one invitation from the same interface, it may then choose to join only one of the groups. The decision can be based on other information carried in the invitation, such as the current group size and the distance to the invitor.
- In case C2 fails to receives an invitation (on some interface) when the TTL value reaches a pre-set threshold, C2 will create a cache group itself and set a timer. C2 can now respond to join requests from others. However if it does not have anyone else join when the timer expires, it will try again to join other groups with an increased TTL threshold. (In the initial deployment when cache servers are rare and far apart, we may need to manually configure the neighboring caches for C2, or have C2 send a message with global scope to WKM to get a list of all caches.)

Using TTL based group discovery favors the creation of groups among caches on the same broadcast LANs or around the same network interconnect point, where the cost of multicasting data is not much higher than that of unicast.

We propose an open membership policy for cache groups. That is, any cache can join nearby cache groups, without going through an authentication step first. Cache consistency and data authentication must be properties that reside in the data, and do not rely on trust of the caches themselves.

Nevertheless, malicious or faulty caches could disturb caching operations by providing false requests or hit reports, or by volunteering to forward a request and not following through. We rely on after-the-fact detection rather that on authentication and pre-screening to identify such disruptive caches. Even the best pre-screening may occasionally fail, making after-the-fact detection a must for all systems.

Group Maintenance

We propose to use an RTCP-like protocol to maintain the cache groups. Each cache in a group multicasts *Group Messages* periodically. The information to be obtained from this exchange includes the group size, the addresses of each cache, and the distance between group members.

The group size and distance information will be useful when the workload for a group is too high (that is, there are too many page requests over short time intervals) and thus the current group must split into two.

When both the workload and the cache hit ratio on a group is too low, the group may consider merging with a neighboring cache group (particularly when some cache is a member of both groups). Suggestions for merging can be communicated via group members, and further information about neighboring groups can be collected to make the merging decision. Merging is done by all members of the current group joining the new group.

Request Forwarding

When a "page miss" in a cache group is detected, some cache or caches in the group must further forward the request towards final resolution. For an individual cache, we need a self-configuring mechanism for that cache to decide if it has a promising neighboring cache, cache group, or outgoing link towards the origin server for forwarding the request. For a cache group as a whole, we need mechanisms to assure that the request with a local miss gets forwarded, as well as to suppress duplicate forwardings.

Where should the request be forwarded?

As we discussed earlier, requests should generally be forwarded "towards" the origin server. However, because caches run on hosts rather than routers, they have no information about the topology of the network. The information a cache C can derive from a request includes (1) the address of the cache N that multicast the request, and (2) the address of the origin

Web server S for the requested page. To make the forwarding decision, C needs to know if it is closer to S than N is, or, less strongly, if it or its neighboring router has an outgoing interface towards S that is different from its outgoing interface towards N. Addressing this question of dynamically determining the request forwarding path will be a central component of our research.

We propose a couple of approaches to this problem. One approach is to make the best use of information that caches already have. For example, if S is on the same network as one of C's interfaces (by comparing the network ID), then C clearly is in a position to forward the request. If we are willing to go one step further to build up such a "forwarding base" for caches, then one can apply a similar approach as the Ethernet bridge learning algorithm, and cache network address information about the direction that answers to requests come from. It might also be possible to add more information to group messages, and have a cache put in the message for group G1 not only all the addresses of its own, but also the addresses of members of group G2, in which the cache is also a member.

A similar approach at a higher level of granularity would be to make use of the "geographical addressing" implicit in the "country" in the domain name. Caches could build up their own forwarding bases of which cache group to ask next for requests for an origin server in a particular country. These forwarding bases could be based both on which caches are nearer to that country, and which caches have had the best past record of answering requests for origin servers in that country.

A second, and complementary, approach is to develop a standard interface to IP routing protocols, so that, for unknown server addresses, a cache can query the neighboring router about that router's output interfaces to the cache N and the origin server S.

It also may not always be the case that the request should be forwarded to a cache group physically closer to the origin server. For example, a small local cache in Australia near the congested trans-continental link might be better off forwarding the request to a large regional or national cache that happens to be in the other direction. Thus, in answering the question of "should I forward the request", in addition to the distance factor a cache may also add in a bias factor that is dynamically adjusted according to the past hit rates for requests sent to neighboring caches.

More research is clearly needed in determining the request forwarding path. Because this decision requires information from routing protocols, it is likely to be the most challenging issue in building our cache design. On the other hand, we should also point out that the request forwarding decision only need to be "roughly right", and the resulting forwarding path is not necessarily "the shortest" one. Take the example in Figure 1 again. Instead of going through the shortest chain of cache groups G1, G2, G3 to reach the origin server group G4, the request from `user-1` may take a longer path through G1, G2, G7, G5 to reach G4, which has little impact on the overall performance. The basic performance gain of the system lies in the caching effectiveness; how fast to get a page the first time is a secondary factor here. Even though a longer forwarding path leads to a longer fetching delay the first time, the performance gain of the system will come from the side-effect of loading up caches, so that subsequent requests for the same page can then be answered with much reduced delay. One way to reduce the worst-case delay for the first request would be to limit the number of “hops” that a request could travel before being forwarded to the origin server.

Which cache should forward the request?

The ideal case would be for exactly one cache in a cache group to volunteer to forward a request. When multiple caches in a cache group volunteer to forward the request, the caches can use a randomized timer algorithm similar to the one in SRM [3] to prevent multiple caches from forwarding the request. However, it is not a problem if occasionally more than one cache in a cache group forwards a request. Duplicate requests are likely to “collide” (run into the same cache group) along the way. In the worst case, duplicate copies of the page may be fetched.

On the other hand, for a request that results in a local miss, it is mandatory that at least one cache in the cache group forward the request. If no caches in the group volunteer to forward the request after a timeout, the cache that brought the request to the group can either contact the origin server directly, or randomly select a cache in the group to forward the request.

Other Issues in Building the Proposed Caching System

This section discusses several issues that are not particular to our proposal, but that need to be addressed by any web-caching infrastructure, including by the manually-configured unicast-based web-caching infrastructure currently being deployed.

How Many Web Pages Are Cachable?

However there have been various concerns about the general feasibility Web caching.

1. One concern is “data correctness”. Without a clear and precise definition of caching functionality, some Web application designers are worried about obsolete data being served to end users. Thus direct dialog between the origin server and client can seem a simple and sure way to do things right.
2. Another resistance to caching comes from commercial content providers who have a vast interest in collecting demographic information regarding page access statistics.
3. There are also concerns about data integrity and copyright issues. The first one needs to verify that the page from the cache is the exact one from the origin server; the second needs to assure that no one but the authenticated client can get a copy.

Because of the above concerns, Web applications are often implemented in a cache-unfriendly way, such as with pages with a lifetime of zero. Such applications are referred to as “cache-busters”.

We believe that adequate lifetime information in each page and strict enforcement by all the caches should address the first concern. For the second concern, we point out that collecting accurate hit counts has only been a goal. Proxy caching is widely deployed today, with no reports back to the origin server on the number of cache hits. We believe that deploying an explicit demographic information collection protocol like the one proposed recently [13] is the best way to collect the needed cache access statistics numbers.

Regarding data integrity and copyright issues, the Internet itself is a store-and-forward system, and nothing can prevent a malicious party from physically modify, or obtaining, a copy of some data that is delivered over the net even without a caching system. The only sure way to achieve data integrity is by applying an end-to-end checksum to detect any modifications. Similarly, the only sure way to control copyright is to encrypt data to prevent wiretappers from seeing the contents.

The question of "how many pages are cachable" translates to the question of "how many pages would be cachable given an effective and globally-deployed web-caching infrastructure". The answer to the second question depends on the attractiveness of the caching system itself. We believe that as the Web caching system is further developed, and it becomes clear that cachable pages are generally delivered to the user faster than are non-cachable pages, more Web applications will be motivated to make their pages cachable. After all, caches serve no purpose but to help the applications get the data to end users faster and more easily.

Caching Effectiveness

Assuming that many or most Web pages are cachable, the next question is how much performance improvement Web caching can bring. The potential gain of caching is three-fold, reducing load at the servers, traffic in the Internet, and fetching delay to the end users. We discuss the first two below.

How much traffic reduction can Web caching bring? The percentage of web traffic out of the total network load gives the upper bound on cache savings. The other limiting factor concerns the cache hit ratios. Recent network traffic measurement numbers suggest that Web traffic has exceeded 50% of the total network load, and this percentage is still growing. However, the relevant question is not how much traffic reduction could web caching bring with current traffic patterns, but how much a ubiquitous and globally-deployed web caching system could contribute to the delivery of current and reliable information to end-users in the long term.

The cache hit rate generally diminishes as the cache moves further away from the original data source. Significant reductions on the server load and on network traffic can only be made by an effective caching system that includes caches near the servers as well as near the end users. Ultimately, the web caching infrastructure needs to be ubiquitous throughout the Internet.

Page Consistency

Most data objects change over time. It is unacceptable to serve obsolete objects from caches without an explicit request from the client to do so. The cache consistency problem is inherent in any caching system, whether it is the current, widely used proxy caching, or our proposal. However, a self-configured web caching system reemphasizes the requirement that the cache consistency and data authentication must be properties that resides in the data, and do not rely on trust of the caches themselves.

Our basic approach to avoid obsolete objects is for caches to enforce lifetime limits on all the cached objects. This can be done with the "opaque validators" in HTTP/1.1 [2]. Each cached object must be deleted when its lifetime expires. If there are no further requests for the information, nothing further needs to be done. If further requests arise, they will bring the latest information to the cache as a side-effect. We can rely on future demand for a page to bring in the most recent page as a by-product.

Incremental Deployment

We plan to collaborate with the Harvest/SQUID Caching team to explore transition strategies to convert the current manually configured caching infrastructure into an autoconfigured, adaptive caching system. This first step would be the incremental deployment into the current unicast caching infrastructure of dynamic mechanisms for forwarding requests to neighboring caches. This is the key next step needed for the current infrastructure to gracefully scale to a larger number of caches. In addition, our proposal would address the incremental deployment of a multicast-based cache architecture into the existing architecture of unicast communications between clients, web caches, and servers.

Summary

We believe that as the Internet becomes more global we must have a self-configuring data dissemination system that can scale with it. We further envision that the basic approaches taken in the web caching infrastructure will be generally applicable to other global-scale information dissemination applications. While there are currently no Internet systems using self-configuration of this nature, we believe that self-configuration is an increasingly-important functionality that will be required by a wide range of Internet systems facing issues of scale.

Comparison with Other Research

Before the invention of the World Wide Web, FTP was the main tool for data dissemination. Heavy loading at popular FTP servers (e.g. the one hosting Internet RFC's) was observed. At that time, however, the network user population was small and the problem was adequately handled by manually configuring one or two replication sites of the same FTP server.

The success of the Web brought unprecedented high demand on Web servers. Facing the overload problem caused by "hot pages", a few measures have been taken recently. One common practice today is proxy caching. Most corporate sites have firewall gateways between their internal network and the public Internet. Web proxy servers are used to relay requests and replies across the firewalls, while at the same time they also serve as caches.

Generally speaking, however, proxy caches do not seem to achieve high hit ratios because they are at the leaves of data distribution trees. Consequently, they do not effectively reduce the load around the origin servers of popular pages. To handle the ever increasing demand for popular pages, some of the busiest Web servers use replication. Manually configured replications may work well for specialized servers with predictable demand, such as the Netscape homepage server, but are not useful in coping with "flash crowds".

The Harvest/SQUID Object Cache is a Web caching infrastructure currently being deployed in the Internet [10]. All cache servers in the SQUID system are connected in a *manually configured* hierarchical tree. As the first step towards reducing unnecessary network load through caching, SQUID has attracted many users, especially overseas network service providers who are concerned with making the most efficient use of the expensive, bandwidth-limited transoceanic links. However, experience has also shown intrinsic limitations of the manual configuration of large systems, such as the burden on system administrators to configure the cache hierarchy and to coordinate with each other, the inevitable human errors, misunderstandings of issues concerning the overall system performance, the desire for local optimization, and the lack of adaptivity to network changes. Australia makes a typical example here: ideally one would like to see all Web caches in Australia group themselves into a cluster which then has one peer connection to the cache hierarchy in the U.S. However, fourteen separate Australian sites configured themselves directly onto the cache tree in U.S. instead of peering with each other locally, leading to the same Web page being fetched directly from the U.S. by each of the fourteen sites. See [11] for more details. The lesson to be learned is that manual configuration of large scale systems is not only burdensome but also vulnerable to errors and misuse. Self-configuring systems, such as the one proposed in this research, can be designed to minimize the possibilities of such abuses.

Furthermore, the single cache hierarchy of SQUID does not provide efficient data routing among all users and servers; it often happens that a new page on an origin server located in California is first fetched by a root node in east coast and then traverses down the cache tree to be delivered to the requester, also located in California. Because all cache misses are fetched by the root nodes first and then disseminate down the tree, the cache hierarchy creates artificial hot spots of cache load near the roots of the tree.

To reduce such overload in a hierarchical cache, Povey has suggested a modification to the SQUID operation [8]. Instead of fetching all new pages through the root nodes, Povey suggested that the hierarchy structure is used for data searching only. When a leaf node, L, searches for a new page and cannot find it anywhere in the tree, the node L itself will fetch the page directly from the origin server, cache it locally, and then send the advertisement of the page up the tree, so that when other nodes search for the same page again they will be able to find it. This modification reduces the load near the top of the tree, but it fails to address the issue of manual configuration, and the need for doing a tree-walk to search for each missing page can also add significant overhead to the system.

Another cache performance study by Gwertxman and Seltzer [5] compares three cache consistency mechanisms currently in use in the Internet: time-to-live fields, client polling, and invalidation protocols. They find that time-to-live fields are a good solution when reducing network bandwidth is the driving force, though client polling are generally a stronger mechanism. There is also a growing literature on caching and removal policies for web caches [9], which we plan to explore during the implementation of our cache protocols.

To facilitate Web caching implementation, the HTTP/1.1 protocol specification provides a number of supporting mechanisms [2]. Server-specified expiration times are added to prevent obsolete data from being served to clients, and a validation mechanism is proposed to eliminate the unnecessary retransmission of previous responses that have not changed. The validation mechanism allows a cache with a long-lived entry to check with the origin server to see if the cached entry is still usable. HTTP/1.1 includes both end-to-end headers that are cachable and hop-by-hop headers that are meaningful only for a single transport-level connection and cannot be cached. Our design will assume the availability of these server-specified expiration times and validation mechanisms. We will also report to the HTTP Working Group any new cache-supporting mechanisms that we discover in our research, so that they can be considered for inclusion in future versions of the HTTP protocol.

To the best of our knowledge, we believe the adaptive Web caching approach outlined herein is the very first proposal to build a *robust, self-configuring* caching infrastructure for global-scale data dissemination. We believe our findings of how to build self-configuring systems will be generally applicable to other loose-coupled, globally distributed systems. To go beyond simulation and lab tests and put our design in global scale real field trial, we have been involved with, and will continue, discussions with the developers of the SQUID cache infrastructure to jointly develop a transition plan to incrementally deploy our protocols, at the proper time, in the SQUID infrastructure.

References

[1] Van Jacobson, "How to Kill the Internet", SIGCOMM '95 Middleware Workshop, August 1995. URL ftp://ftp.ee.lbl.gov/talks/vj-webflame.ps.Z

[2] R. Fielding, J. Gettys, J. Mogul, M. Frystyk, T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", Internet Proposed Standard protocol, RFC-2068 URL ftp://ds.internic.net/rfc/rfc2068.txt.

[3] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steve McCanne and Lixia Zhang. "A Reliable Multicast Framework for Lightweight Session and Application Layer Framing". *Proceeding of ACM SIGCOMM '95.* Aug. 1995.

[4] Sally Floyd and Van Jacobson, Random Early Detection gateways for Congestion Avoidance, IEEE/ACM Transactions on Networking, V.1 N.4, August 1993, p. 397-413.

[5] James Gwertxman and Margo Seltzer, "World-Wide Web Cache Consistency", USENIX 1996, URL http://www.eecs.harvard.edu/ vino/web/usenix.196/.

[6] Margo Seltzer, "The World Wide Web: Issues and Challenges", Presented at IBM Almaden, July 1996, http://www.eecs.harvard.edu/ margo/slides/www.html

[7] Venkata Padmanabhan and Jeffrey Mogul, "Improving HTTP Latency", URL http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html.

[8] Dean Povey and John Harrison, "A Distributed Internet Cache", 20th Australian Computer Science Conference, Sydney, Australia, February 5-7. 1997, URL URL http://www.isi.edu/lsam/ib/http-perf/.

[9] Stephen Williams, Marc Abrams, Charles Standridge, Ghaleb Abdulla, and Edward Fox, "Removal Policies in Network Caches for World-Wide Web Documents", Sigcomm 1996.

[10] "A Distributed Testbed for National Information Provisioning", URL http://www.nlanr.net/Cache/.

[11] Duane Wessels and Kim Claffy, "Evolution of the NLANR Cache Hierarchy: Global Configuration Challenges", November 1996, URL http://www.nlanr.net/Papers/Cache96/

[12] Jun Li, Si Yuan Tong, and Adam Rosenstein, "MASH: The Multicasting Archie Server Hierarchy", Project report, December 1996, UCLA Computer Science Department.

[13] Jeffrey Mogul and Paul Leach, "Simple Hit-Metering for HTTP", Internet Draft, January 1997.

[14] L. Zhang, S. Deering, D. Estrin, S. Shenker, D. Zappala "RSVP: A New Resource ReSerVation Protocol", IEEE Network, September, 1993

[15] R. Bagrodia and W. Liao, "MAISIE: A Language for the Design of Efficient Discrete-Event Simulations", IEEE Transactions on Software Engineering, Vol. 20(4), April 1994, pp. 225-238.

[16] "Virtual InterNetwork Testbed", URL http://netweb.usc.edu/vint/