Application of Distributed Web Site Acceleration: Mars Polar Lander

Niall Doherty with contributions from Anil Madhavapeddy | Network Appliance | July 2000 | TR 3071

TECHNICAL REPORT

Table of Contents

[TR3071]

---

0. Abstract

This document is aimed at those interested in the technical aspects of hosting content on the internet while designing for cacheability. It explains some of the concepts involved and describes the approach taken to use them in the deployment of the *www.marspolarlander.com* distributed Web site. After studying this document, the reader should:

- realize the benefits gained from designing for cacheability,
- see why using distributed Web sites augments this, and
- understand how to do the above.

1. Architecture and Deployment

1.1 Introduction

Data from the surface of Mars is received by the Deep Space Network (DSN), which is managed by the Jet Propulsion Laboratory (JPL) for NASA. Some of this information is transferred to the University of California at Los Angeles (UCLA), via Internet 2 links, for further processing by the Mars Volatiles and Climate Surveyor (MVACS) group.

Images will make up a significant proportion of the data, and during busy periods it is estimated that one 256-bit x 256-bit image could be received by UCLA every minute. When these *raw* images are received they will be converted into the Internet standard GIF and/or JPEG formats and made immediately available to the public by being uploaded automatically to the assigned Web site. Occasionally, higher quality images are to be produced (full color, etc.).

It is expected that the images portion of the data will be extremely popular with the public, and it is desired to deploy Web accelerator servers at strategic points on the Internet to handle high loads and direct traffic away from the UCLA origin site [HOSTING].

1.2. System Overview

Basic Components Involved in a Distributed Web Site

The following components are all involved in the HTTP transactions that occur *(see Figure 1)*:

- Origin Server - *(controlled)*
  - Content is generated and stored here.
  - Only accelerators should request content from this server.
- Accelerators - *(controlled)*
  - Handle much higher load than standard Web servers.
  - Placed at strategic locations; deployed to bring specific content closer to end users.
  - Load balancing DNS software ensures end-user requests are directed to the closest accelerator.
- Caches - *(uncontrolled)*
  - Caching of Internet objects is now widespread throughout the world.
  - Many ISPs and national telecoms have deployed Internet Caching Servers.
- Clients - *(uncontrolled)*
  - Many browsers implement local (private) caches following standard HTTP rules.

Note that although physical control of private and shared caches worldwide is not possible, it **is** possible to exert some control over how they cache data drawn from an origin site. It is important, therefore, to view all of the above components as part of the “distributed Web site.” Designing for cacheability is wise even if acceleration is not being used!

1.3 Partners and Roles

Partner	**Role**
Network Appliance	Provide caching/accelerator appliances (NetCache®).
F5 Networks	Provide load-balancing DNS servers (3DNS Controller).
Cable and Wireless, USA	Provide hosting locations at various POPs.

1.4. Deployment Configuration

Since there is no longer any significant load for this Web site, the accelerator network has since been dismantled.

NetCache accelerators are located at the following sites:

New York City, NY
San Francisco, CA
Reston, VA
Chicago, IL

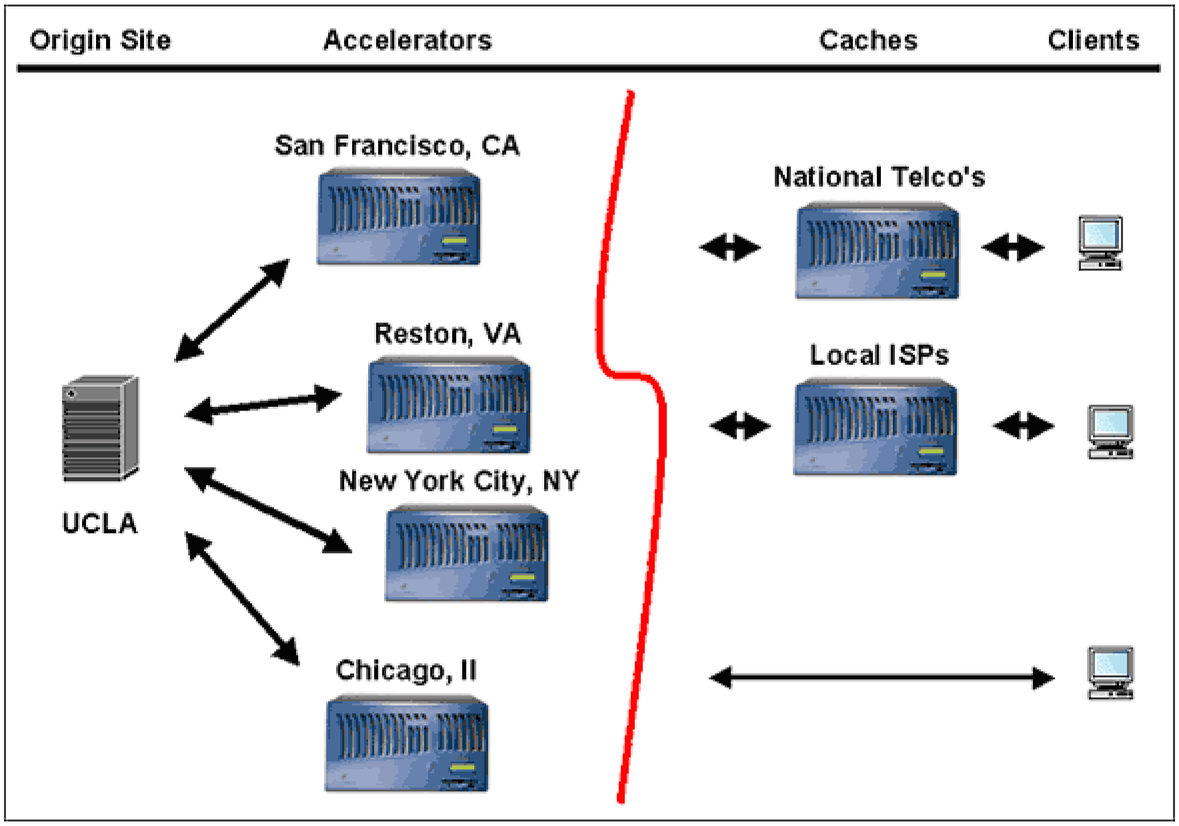

Figure 1: Interaction between clients and distributed Web site.

For the Mars Polar Lander Web site, Global Server Load Balancing [Section 4] was implemented using the 3DNS Controller from F5 Networks. One of these was co-located with each NetCache accelerator. The devices implement a GSLB name server and also include a proximity measurement agent that actively measures distances to client networks.

The *marspolarlander.com* domain was delegated to the hosting partner and managed by the 3DNS Controllers. These controllers answered queries for *www.marspolarlander.com*, returning the IP address of one of the accelerators. A weighted combination of the hop count and roundtrip time algorithms was used to choose the best pathway through the network.

[See Appendix A for network configuration details.]

2. HTTP and Caching Mechanisms

HTTP [RFC2616] operates using a client-server mechanism where a server replies in response to a client query. Both replies and responses consist of *headers* and *data*. Servers (and not clients) can use headers to control whether or not, and how, the data can be cached (both by clients and shared caching servers).

2.1. Benefits of Caching

Caching technology is used for two main reasons:

1. Reduction of Bandwidth Requirements
2. Response Time Improvements

Both of these arise because objects are brought closer to end users, serving them faster and reducing the need to request them from origin servers so often.

A **Web accelerator** is simply a cache that is dedicated to a select number of Web sites. Accelerators offload traffic from both:

1. the origin Web servers (traditional deployment), and
2. the origin network (distributed deployment).

Note that accelerators should not be a replacement for good cacheability design.

2.2. Hits, Misses, and Validation

A cache may store many objects, but it is important to know which of these objects are still valid:

- Validation is the process by which a cache contacts an origin server to verify whether an object that was retrieved from that server is still valid (i.e., not *stale*).
- A validator is an element (typically included in HTTP response headers) used to find out whether a cached object is an equivalent copy of an object on the server (e.g., *Last-Modified* time).

Requests received by caches can be broadly classified as follows:

- **MISSes**
    - The object is not present in the cache and must be retrieved from the origin site.
    - Response Time is similar to when no cache is deployed.
- **HITs Requiring Validation**
    - The object is present in the cache, but must be validated with the origin server.
    - Response Time is poor because the origin server must be contacted. However, this will be improved somewhat over the case of a MISS as, frequently, the object will not have been modified and will not need to be transferred again.
    - Bandwidth will be saved if the object has not been modified.
- **HITs Not Requiring Validation**
    - The object is present in the cache **and is known to be valid**.
    - Response Time is very fast because the origin server does not need to be contacted.
    - Bandwidth reduction is optimum.

It can be seen that it is very important to ensure that as little validation as possible is required. This will be achieved if the origin server assigns explicit expiration times to objects in the HTTP *response* headers. The following shows two possibilities:

```
Expires: Wed, 03 Nov 1999 00:45:48 GMT

Cache-Control: max-age=86000
```

Note:

- *Cache-Control: max-age* is preferred over *Expires*.
- In general, an object will **not** be cached if neither a validator nor an explicit expiration time is present.

2.3. "Dynamic" and "Static" Content

Caching of "Dynamic" Content

The HTTP specification does not distinguish between "dynamic" and "static" content. Content cacheability is decided entirely by the HTTP headers that are returned in the response from the origin server; the method by which the content is generated is irrelevant.

Much content that is generated "dynamically" by a server (e.g., database queries) is valid for some period of time and, therefore, should be cacheable. It is generally straightforward to design the server so that this content can be cached. For example, a "script" that queries a database can send an explicit expiration header (typically after a *Content-Type* header) with the response. Of course, some content must always be retrieved from the origin server (e.g., bid prices for an online auction); in this case, an appropriate *Cache-Control* header should indicate this fact.

Note that making a URL cacheable even for only a short period can have dramatic effects. For example, if a particular URL is requested 100 times per minute, making this cacheable for 1 minute could remove 99% of the load from the origin server if an accelerator is being used.

Finally, it should be obvious that the majority of bytes (typically 60% or higher) from Web sites are "static" (e.g., images, Java applets). Even though the HTML may be generated anew for every request, many items within the page may possibly be cached.

Historical Note

Caching servers were first developed during a period when the HTTP specification (1.0) did not explicitly discuss caching techniques. Web servers could not exert explicit control over content and it was not uncommon for caching server heuristics to cause content to be cached for too long a period. This was a particular problem with dynamically generated content, which changed frequently because the server had no way to control how long caching servers should cache it.

To work around this problem the first caching servers introduced the concept of a "stoplist". This is a list of URLs (specified using regular expressions) that should not be cached by the caching server. The default configuration usually specified URLs including "cgi" and "?" to be uncacheable.

HTTP 1.1 now allows for explicit control of content and caching servers, in general, do not cache content if there are no appropriate headers. The stoplist is, therefore, somewhat redundant. However, many caching servers will probably include a preconfigured stoplist as part of the default configuration for some time.

2.4. Cache Content Control

To Cache or Not To Cache?

If an origin server does not want an object to be cached under any circumstances it should send the following HTTP *response* header:

```
Cache-Control: no-cache
```

If an object is specific to a particular enduser, it may be feasible to allow it to be cached by a client but not by shared caches. The following HTTP *response* header can be used for this purpose:

```
Cache-Control: private
```

It is possible for a client to request an object and specifically indicate that it does not want a cached copy. In this case, it should send the following HTTP *request* header:

```
Cache-Control: no-cache
```

Forcing Revalidation of Objects

To **force** a URL to be revalidated, the following is required:

```
Cache-Control: must-revalidate
```

The HTTP/1.1 specification states that an object does not need to be revalidated if it is still fresh, so an explicit expiration time should be assigned. This needs to be set so that the object is considered stale immediately.

This will also override any NetCache default Time-To-Live (TTL) that is assigned (see Appendix B *for more details).*

```
Cache-Control: must-revalidate, max-age=0
```

However, if a Last-Modified header is not present, the *max-age=0* may cause the object not to be cached, so the following should force it to be cached and revalidated on every request:

```
Cache-Control: must-revalidate, public, max-age=0
```

Prefetching

In general, when a new object is placed onto an origin site, caches will not retrieve this object until a client makes a query for this object.

Network Appliance has implemented some simple features that can be used to control explicitly the content stored in its caches. It is possible to *delete* objects from and *preload* objects to NetCache using *extended* HTTP request headers:

```
Cache-Control: eject

Cache-Control: prefetch
```

It is relatively straightforward to develop a simple [client-side] script that will implement this functionality. These “commands” are simply HTTP headers that are sent as part of a “normal” client request, e.g.:

```
GET /index.html HTTP/1.1
Host: www.x.com
Cache-Control: prefetch
```

Note that the *prefetch* command will be treated as a normal client request and, thus will be subject to normal caching rules (validation, etc.). To ensure that an object is fetched “fresh” from the origin server, the following header should be used:

```
Cache-Control: no-cache, prefetch
```

[Note that there is currently a NetCache bug which causes this to fail on occasion. To ensure the correct behavior it is recommended to send an eject *command, followed by a* prefetch *command.]*

It is important to realize that, in practice, preloading content into a cache only improves the performance for the *first* user to retrieve the object. After the first request it will be in the cache, regardless of the method of retrieval, and will be treated like all other objects for subsequent requests.

2.5. Cookies

General

A cookie is an identifier used to maintain state information across HTTP sessions. Cookies allow servers to (among other things):

- “remember” individual user preferences and/or profiles, and
- track user browsing within a site.

A server “sends a cookie” to a client by including a *Set-Cookie* HTTP header in a response. This instructs the client to modify an existing cookie or, if it does not exist, to create it. A cookie contains key/value pairs, including:

- *Identifier* - allows tracking
- *Expires* - expiration information
- *Domain* - the DNS domain names for which it is valid
- *Path* - URL paths in the domain for which it is valid
- Optional variables (and values)

[The majority of browsers allow the handling of cookies to be disabled or restricted in some manner. However, it is generally accepted that there are few security risks with the use of cookies.]

Clients will send *Cookie* HTTP *request* headers when their requests match the *domain* and *path* parameters for any cookies they have currently stored. For example, if a server sent a cookie with a path of “/”, the client should send a *Cookie* header in every request to that site. The client will include all the appropriate key/value pairs as parameters to the cookie request header.

Caching and Cookies

Responses from servers containing cookie headers are assumed to be intended for a particular client and should, in general, not be cached by caching servers. However, it is not unknown for servers to send cookies with many (or all) of their objects. This may be through a lack of understanding (or interest) of cookie behavior or, possibly, a deliberate attempt to make their content uncacheable (obviously, the appropriate *Cache-Control* headers should be used instead).

NetCache provides a configurable list—blank, by default—that allows the end user to specify that for URLs ending with specific “extensions” any *Set-Cookie* headers in the response will be ignored and the object will be subject to normal caching rules.

In short:

A **response** with a *Set-Cookie* header will not be cached, unless

- a *Cache-Control: public* header is present, or
- the URL matches an item on our Cookie Cheat Extension list.

A **request** with a *Cookie* header will be passed through to the origin server unchanged. The response will be treated as any other response would be.

Cookies should only be sent by servers when necessary. For example, if it is desired to track a user's activity on a site, it is not necessary to send cookies for every response. If the *domain* and *path* are set correctly when the user first “enters” the site, the cookie request headers from the client can be recorded—there is no need to send cookies for every object.

3. Web site Acceleration

3.1. Client URL Requests

A client must initiate a connection with a server to request a URL. If the URL contains an IP address as the hostname it can connect immediately. If not, it must issue a *DNS query* to a DNS server to have the hostname converted to an IP address.

The Domain Name System (DNS) [DNS1998] is a worldwide distributed database responsible for, among other things, mail routing and converting human-readable names, e.g., *www.x.com*, into Internet addresses (IP addresses), e.g., 1.2.3.4.

The general format of a URL [RFC1738] is as follows:

```
protocol://[username:password@]hostname[:port][url-path]
```

Examples of URLs are:

```
ftp://john:trythis1@ftp.x.com/dir/file.gif
http://www.x.com
http://10.20.12.5/index.html
```

3.2. Server Request Handling

The initial HTTP specification allowed requests of the form:

```
GET url-path HTTP/1.0
```

The concept of multiple Web sites being managed by one server was not provided for, which became desirable when service providers began to host Web sites for many customers. Using one server per Web site was wasteful. It was possible to work around the problem by noting that each HTTP connection between client and server is completely described by:

```
{client IP address, client port, server IP address, server port}
```

Web servers were then designed to take advantage of this fact by supporting multiple Web sites and using the destination IP address in the client request to distinguish between them.

The Web server would be configured to accept multiple IP addresses, with each IP address corresponding to a different Web site. This would be transparent for clients since the DNS names of the Web sites would all be different and each would map to one of the IP addresses supported on the Web server.

While this provided a solution to work around the limitations of the protocol, it required (wasted) many IP addresses.

The HTTP/1.1 specification resolves the above issue by building into the protocol the ability for a client to request a URL and also to specify which Web site it desires the URL to be retrieved from. The mechanism for this is the *Host* header. A client request now looks like this:

```
GET url-path HTTP/1.1
Host: www.x.com
```

A Web server now only requires a single IP address and may serve many Web sites, choosing the appropriate Web site based on the *Host* header in the client request. *Note that there are, however, many Web sites still using the old legacy method.*

3.3. Cache Servers as Web site Accelerators

If a cache server is to be used as a Web site accelerator, it is necessary to ensure that clients connect to it—and not the origin server—when they request URLs from the Web site.

Transparent Caching and Server Load Balancing

The simplest deployment is to implement transparent caching so that requests to the origin server are redirected to the cache server. This deployment is only possible if the cache server and the origin server are located “close” together. Note that the cache server is operating in normal “forward proxy” mode.

Typically, busy Web sites will use some device to implement server load balancing across a number of origin servers. Some of these devices also provide transparent redirection facilities. In this case, a single device could implement server load balancing **and** also redirect client requests, as they arrive, to the cache server(s).

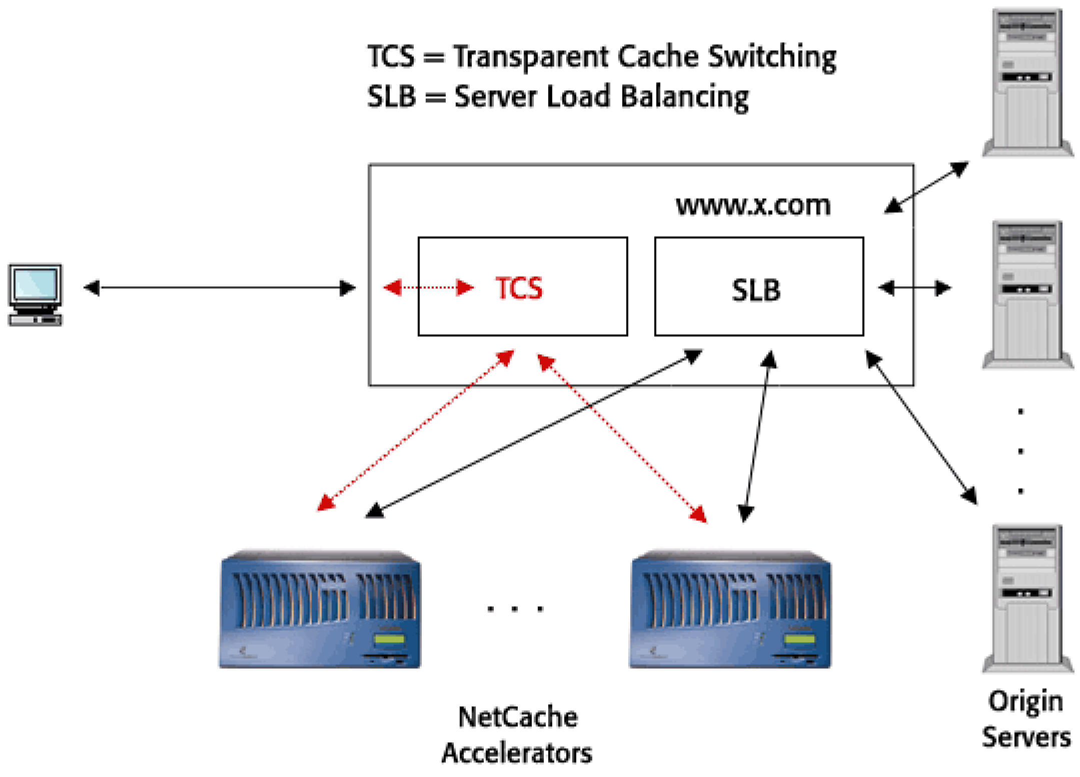
TCS = Transparent Cache Switching
SLB = Server Load Balancing
www.x.com
TCS
SLB
NetCache Accelerators
Origin Servers

Figure 2: Web Acceleration utilizing Transparent Caching

DNS "Routing"

Another approach is required when the accelerator and the origin server are not located "close" together. The most common method is to use DNS to direct client requests appropriately: when a DNS query is issued to find the IP address of the hostname for a URL the DNS server is configured to respond with the IP address of the accelerator. This means that when the client initiates a connection, it does so with the accelerator rather than the origin server. The accelerator is configured to forward requests for [as yet] uncached objects to the origin server.

(Note that the DNS servers are typically controlled by the same organization responsible for the Web accelerators.)

To implement Web site acceleration it is necessary to perform a mapping function between client requests and server requests. A request for an accelerated object must be sent to an appropriate origin server **and** to an appropriate Web site if multiple Web sites are being handled.

The output of the function is:

- the origin server and port number that should be connected to, and
- the *Host* header to use in the request to the origin server.

The input is either:

- the destination IP address of the client request
  (the accelerator would be configured to accept multiple IP addresses), or

- the client *Host* header
  (preferred method).

As a simple example, consider an accelerated Web site known as *www.x.com (see Figure 3)*. The following configuration could be used:

- The DNS server for the *x.com* domain is configured so that:
  - A query for *www.x.com* gives the IP address of the accelerator.
  - A query for *www-origin.x.com* gives the IP address of the origin server.
- The accelerator is configured to use *www-origin.x.com* as the origin server.

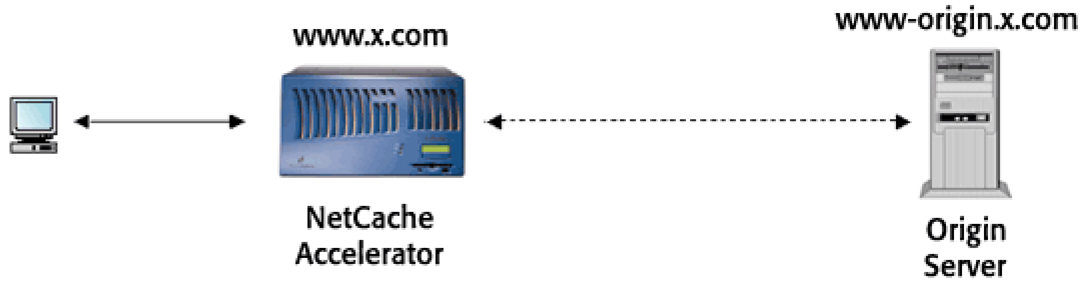

Figure 3: Simple Web acceleration deployment.

The following shows the relevant headers in the client request:

```
GET url-path HTTP/1.1
Host: www.x.com
```

The relevant headers in the request from the accelerator to the origin server are shown below:

```
GET url-path HTTP/1.1
Host: www-origin.x.com
```

Note that the *Host* header has been rewritten. If this is not done, a problem could occur if there is a transparent cache server located “between” the accelerator and the origin server.

[Cache servers typically use the Host *header to determine which Web site to contact by performing a DNS query for its IP address.]*

If a *Location* header is returned in the HTTP response from the origin server, this should be rewritten, if necessary, to ensure that the accelerator is referred to and not the origin server.

If it is desired to use the same accelerator to accelerate the Web site *www.y.com*, then the same procedure as above would be followed. In particular, note that a DNS query for *www.y.com* would give the same resulting IP address as a DNS query for *www.x.com*. The client *Host* header allows the accelerator to distinguish between requests for different Web sites, and the mapping function will cause the accelerator to retrieve the appropriate responses.

Redundancy and Server Load Balancing

A common configuration is to have the DNS query for the web site return a virtual IP address on a “switching” device. This device can then load balance between multiple accelerators.

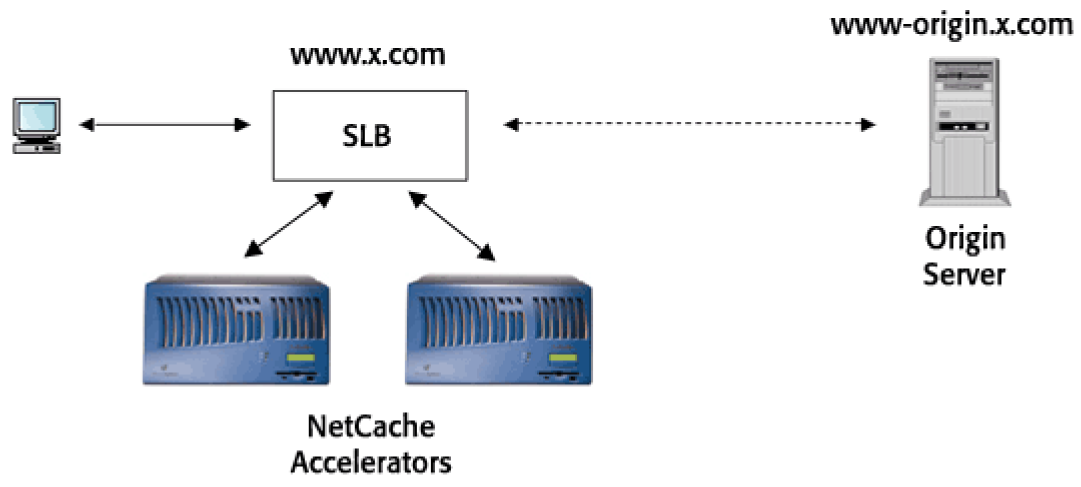

Figure 4: Server load balancing with a distributed Web site.

Another possibility is to use the device to failover to the origin server if the accelerators fail to respond. There are a number of possible approaches here. The device could issue a HTTP redirect to the origin server. Alternatively it could send the request straight through to the origin server. In the latter case, note that the origin server would need to be configured to recognise *www.x.com* and *www-origin.x.com* as being the same Web site.

4. Global Server Load Balancing (GSLB)

4.1. General

There are a number of different approaches to Global Server Load Balancing (GSLB) available. The most common is a DNS-based system. Other alternatives are available, however, such as "Global IP."

This section discusses the components of these systems and how they interact.

Node

A node is a distinct location on the network where there are one or more servers that are part of the distributed Web site. A single server may be sufficient in some situations. If more significant load is expected, then a number of servers may be placed at the node and some device can be used to load balance traffic between them.

If there is a single server, then the address of that node will be that of the server. With multiple servers, typically, the address will be that of a Virtual IP Address at that node that is being used for "local" server load balancing.

Note that some Web sites replicate the origin server(s) at another physical location for redundancy. It is possible to include the origin server locations in the nodes that are globally load balanced, but for the purposes of this discussion it is assumed, for simplicity, that all nodes consist of accelerators—and not origin servers.

4.2. DNS-Based GSLB

GSLB Name Server

The name server is responsible for distributing the client requests among the nodes. When a client requests the IP address of a hostname that is being globally load balanced, the name server replies with the address of the closest node.

The name server typically maintains and builds a table matching clients (or, more likely, client **subnets**) with the closest nodes. This table is constantly updated as new information is received from the nodes. The Time-To-Live (TTL) that the name server assigns to its replies is usually quite short so that changes in the network can be propagated rapidly.

Various criteria are used to make the decisions. Some devices allow these to be selected and arranged in different priorities. Some typical criteria are:

- health of accelerators at each node
- capacity of load balancing devices at each node
- geographical location of the clients (typically, by continent)
- client-node proximity

Proximity Measurement Agent

Some “intelligence” is required at each node if proximity between the client and the node is to be measured. Some agent, therefore, is located at each node to achieve this function. Typically, a device that performs load balancing also performs proximity measurements.

The agent may make **active** measurements to a client's network. In this case, when the name server receives a request it sends a command to the agents and instructs them to begin measuring the “distance” to that client. Methods similar to “pings” or “traceroutes” are used with such information as Round Trip Time (RTT), hop count, and packet loss being noted. Although this method will suffice in many circumstances, there are a number of limitations. The name server actually receives the request from the client's local name server—not the client itself—so measurements are only possible to the local name server. Also, it is not uncommon for some of these methods to fail, as they may be blocked by security mechanisms.

Another (possibly superior) approach is to use the **natural traffic flow** between the client and the node to measure the round-trip latency. This could be done by a device which can be used for both load balancing and proximity measurements. Typically, the TCP handshakes between the client and the node's [virtual] IP address can be observed and used for an appropriate measurement. By directing a small percentage of requests to different nodes at random (subject to other criteria such as node health etc.) the device can gain an accurate view of the dynamic network conditions and, over time, more requests will be directed consistently.

Figure 5 shows the various components involved in the deployment.

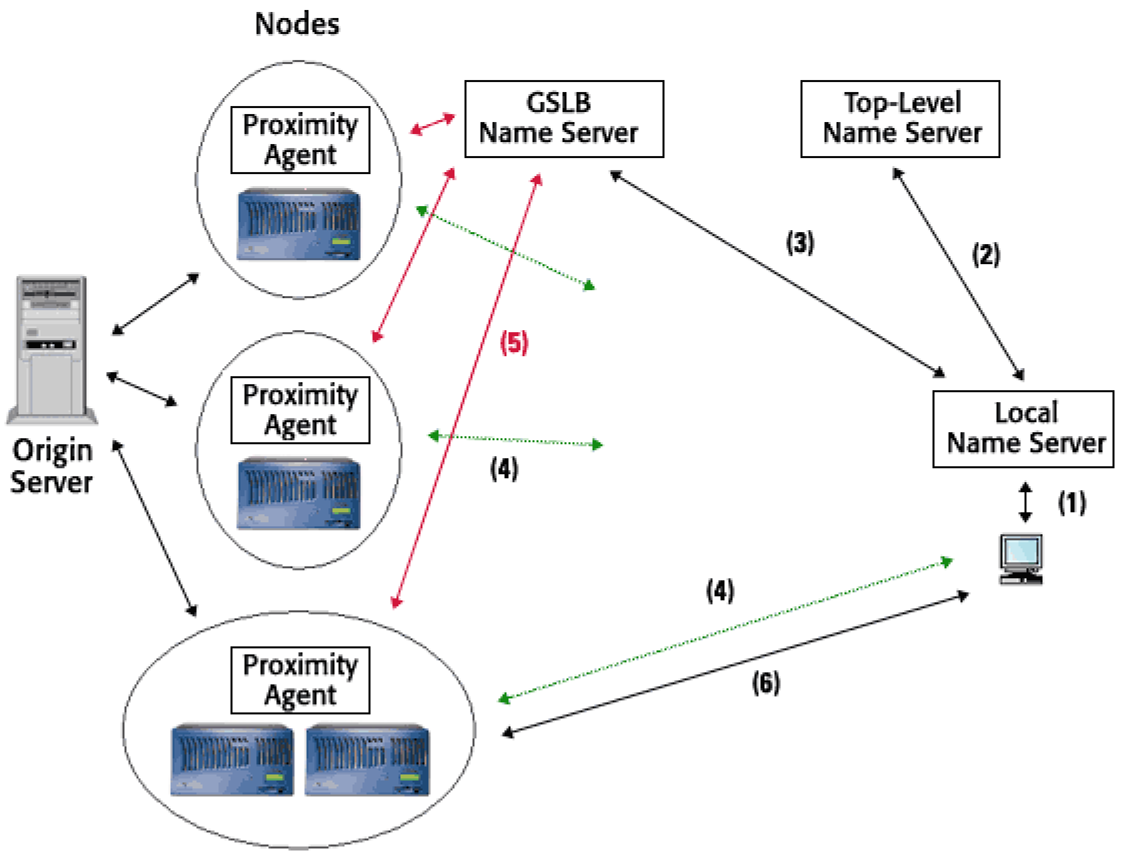

Figure 5: DNS-Based Global Server Load Balancing.
The client queries its local name server for an IP address of globally load-balanced hostname (1). The local name server queries appropriate name servers (2) and finally queries the authoritative (GSLB) name server (3), receiving a result that is based on proximity measurements (4) between its local network and the various nodes. The nodes communicate proximity measurement results to the authoritative name server (5). The client communication to the node (6) may be monitored for further measurement data.

4.3. Global IP

This approach relies on network-level routing to distribute traffic. All nodes are assigned the *same IP address*. The nodes are monitored for health statistics by some network device and, if the node is "healthy," a host route is injected into an adjacent router. Client traffic attempting to reach the Web site will have a choice of paths to the destination. The path chosen will be based on route path costs and will typically take the shortest distance.

Note that a number of vendors provide equipment with the functionality discussed in this section, e.g., Foundry Networks [FNDRY], Alteon WebSystems, and F5 Networks.

5. Web site Content Management Issues

5.1. Extended Cache-Control Functionality

NetCache 4.0 introduced a new header:

```
  X-Accel-Cache-Control:
```
This header takes the same options as *Cache-Control* but differs from it in the following ways:

- It takes priority over any *Cache-Control* header.
- It is removed from the headers as soon as it is processed by the first NetCache that encounters it.

This functionality:

- allows accelerators to receive cache control commands different from other caches, and
- ensures that “specialized” cache control commands, such as *eject* and *prefetch*, are not sent to an origin server when included in a “client” request.

As an example of its use, the following shows the header that an origin server could send to an accelerator to instruct it to preload an image:

```
  X-Accel-Cache-Control: no-cache, prefetch
```

If both an *X-Accel-Cache-Control* and *Cache-Control* header are sent in a reply, the former will take precedence for an accelerator, but the latter will be stored and sent to any client or cache that requests the object.

There is one issue to be aware of, however. Since the *X-Accel-Cache-Control* header is stripped by the first NetCache that encounters it, it is possible that a cache (e.g., a transparent cache) along the HTTP path between the origin site and the accelerator could unintentionally “intercept” the request, and thus the command intended for the accelerator will not reach the desired destination. This is not a difficulty when the origin site and accelerator are co-located, but it may be an issue when distributed accelerators are deployed.

5.2 Accurate Log File Analysis

Many content providers find it necessary to record access to their Web site in part if not in full. There are many methods of doing this, but careful consideration is required if the site is being designed with cacheability in mind.

Because caching is now wide spread, objects can be cached throughout the Internet and client accesses may not always cause log entries to be made on the origin site. It is, therefore, not meaningful to rely on origin server logs as an accurate assessment of an object's popularity unless special arrangements are made.

Many Web sites have used a simple, but inelegant, method of guaranteeing origin site log entries: the site is made uncacheable by using, for example, *cookies* or *Cache-Control: no-cache*. This achieves their goal, but there are better ways to accomplish this.

It should be noted that a “Web page” generally consists of a number of objects (URLs). For example, the following Web page consists of three objects (one HTML URL and two GIF URLs):

```
  <HTML>

  <IMG src="pic1.gif">
  ...
  <IMG src="pic2.gif">
```

```
</HTML>
```

To record access to this page, only log entries for the HTML URL are required; any “embedded” images can be cached as normal. Instead of making the page uncacheable, however, it is possible to force it to be *revalidated* on every access, but still be cacheable!

```
Cache-Control: must-revalidate, public, max-age=0
```

It may be difficult to create the required control headers for all the various pages that require counting, but there is an alternative approach that can be used. A very small item can be created and all pages that require counting can embed this in them. Accesses to the item can be traced back to the “embedding” page by examining the *referrer* log entries. A good choice for the item could be a 1-bit x 1-bit invisible transparent GIF image. A logo may also suffice.

The above *Cache-Control* statement, when applied to a particular object, will cause NetCache 4.0 (and other web caches that obey the HTTP/1.1 specifications) to cache that object when retrieved **and** revalidate it with the origin server for every access! This implies that every request for the object will be logged on the origin server. If the object has not been modified, the server will return a *Not-Modified* response, the connection will close immediately, and the cache will serve the object from the local store.

To summarize, recording accesses to web pages can be achieved by:

- Embedding the web pages to be counted with a small transparent GIF image.
- Ensuring the image is cached but always revalidated.
- Using the referrer log information to trace the image access back to the referring page.

This design gives accurate reporting of accesses while maximizing the cacheability of the objects. However, revalidation is taking place, so it is best to use this only on “top-level” or important Web pages.

Rather than recording all accesses on the origin site, it is possible to instruct caches to revalidate an object on every request, while allowing accelerators to cache it. Since each accelerator is controlled by the content provider, the logs from the accelerators and the origin site can be concatenated. In this case, the origin server could send the following HTTP response headers:

```
X-Accel-Cache-Control: max-age=86000
Cache-Control: must-revalidate, public, max-age=0
```

5.3 "Pushing" of Content

As discussed earlier, if a web site is designed for cacheability then *preloading* or *pushing* of content does not lend any great benefits. However, there are some cases when it may be useful. Consider the situations where it is not known how often particular URLs may change or where it is difficult to control the expiration headers in detail (possibly for non-technical reasons).

A common approach with some implementations of "Content Distribution Networks" (CDNs) is to run some "agent" that monitors content on the origin server and, when changes are detected, pushes this new content out to the accelerators at the "edges" of the network.

Obviously, it is not possible to contact every caching server that may have copies of old versions of this content so the only possible approach is to ensure that caching servers do not consider the content they retrieve as always being valid. This could be achieved if the agent assigned headers to the objects so that the objects would be either uncacheable or, preferably, revalidated on every request by normal caching servers.

Since the accelerators would have the new content uploaded to them when it changes, they can be instructed to cache the content for an arbitrarily long time. The following headers would allow the above criteria to be met:

```
  X-Accel-Cache-Control: max-age=31449600
  Cache-Control: must-revalidate, public, max-age=0
```

Note that the above approach implies that more content will be served from accelerators than if the web site was designed for cacheability - and many CDNs charge per byte served from their accelerator network. Designing the web site for cacheability may be a more elegant approach.

Comment on Streaming Media On-Demand Content

Many caching servers now serve streaming media content.

If a client requests a stream from a distant origin server it is likely that only a low quality stream will be received because of bandwidth constraints. This implies that, if caching servers can only retrieve content by proxying client requests, it is unlikely that high quality streams will be cached.

However, if a cache could be designed to receive streaming media data independent of client requests and store it in an appropriate manner, then it would be possible to "push" high quality streams out to the edge of the networks where clients can now access them with higher bandwidth capacities.

There is, therefore, a fundamental Quality of Service issue that can be addressed with preloading Streaming Media content - there is no equivalent issue for HTTP content.

6. Design of Mars Polar Lander Origin Site

A simple design, with attention given to cacheability, results in dramatic Web site performance increases.

6.1. Servers

Functions of Origin Servers

The MVACS origin site consists of a number of different servers:

static.mars.ucla.edu (www.mars.ucla.edu, mvacsweb4.mars.ucla.edu)

This server holds all the “static” Web pages (some of which contain Javascript embedded within them) and links to the other servers. No content on this server is generated dynamically.

live.mars.ucla.edu (mvacsweb3.mars.ucla.edu)

“Live” data received from Mars is routed straight to UCLA for processing. Software on this server analyzes the incoming data from the Ground Data System (GDS) and generates “static” HTML and data (e.g., images) files.

The following systems are handled:

- Surface Stereo Imager (SSI)
- Robotic Arm Camera (RAC)
- Mosaic Creation (incoming images are sized, balanced, and inserted into a mosaic)

products.mars.ucla.edu (mvacsweb5.mars.ucla.edu)

The various science teams assemble *high-order products* from the raw instrument data transmitted by the lander. These take many forms, from mosaics to stereo anaglyphs and even executable programs.

The data is uploaded to this server via an internal operations submittal form. The product(s) are then encapsulated in an HTML page or other appropriate Web delivery presentation.

backup.mars.ucla.edu (mvacsweb2.mars.ucla.edu)

This machine receives *tar*ed and *gzip*ed data from each of the other machines for permanent archival.

Hardware and Software

Each of the origin servers uses dual Pentium II 450 CPUs. They are equipped with 512MB RAM and approximately 20GB of internal storage. An external RAID5 system provides an additional 50GB of storage. The Operating System (OS) used is Solaris™ x86.

The Apache open-source Web server was chosen for its flexibility, reliability, and stability. The embedded scripting language PHP is used for "active content" pages, while the Perl scripting language is used to generate "static" HTML pages. A MySQL database stores all the high-order products that are created.

6.2. Creating the Appearance of Single Web server

The Web site is configured so that the content appears as if hosted by a single Web server. This is achieved by using an internal proxy and rewriting module of the Web server that allows mapping of remote servers into the local namespace:

- The *www.marspolarlander.com* site accelerates the *www.mars.ucla.edu* origin server.
- The "live" and "products" content are both referred to by the following URLs:
  - `http://www.marspolarlander.com/l/`
  - `http://www.marspolarlander.com/p/`
- The *www.mars.ucla.edu* server is configured to rewrite requests internally:

Location	Maps To
`/l/`	`http://live.mars.ucla.edu/`
`/p/`	`http://products.mars.ucla.edu/`

- When a request is received by *www.mars.ucla.edu* for "live" or "products" content the request is internally rewritten and proxied to the appropriate server. The response is received by *www.mars.ucla.edu* and sent to the client as if generated locally.
- The result is as follows:

Origin Content Location	Referenced By
`http://static.mars.ucla.edu/`	`http://www.marspolarlander.com/`
`http://live.mars.ucla.edu/`	`http://www.marspolarlander.com/l/`
`http://products.mars.ucla.edu/`	`http://www.marspolarlander.com/p/`

6.3. Cache Content Control

The details described below were not finalized, and would have been modified slightly over time to achieve the best performance. Since the Mars Polar Lander was, unfortunately, lost during atmospheric entry, no further work was done.

Overview

The site contains both static files and dynamically generated content. Since cacheability of the site is a prime concern, the majority of the "dynamic" data is actually pregenerated and stored in static files. Care is taken to ensure that each item is referenced with a unique URL so that future references to this same object will result in the same data, allowing it to be cached reliably.

The origin servers use expiration times for their content that depend on how often the target pages will change. For example, the news and updates pages have low expiration times (1 or 2 hours), while informational sections, such as instrument descriptions, have longer expiration times (weeks or more). Objects with low expiration times are designed to be served very rapidly (e.g., no server-side parsing is required) so that the accelerators can retrieve and store the pages as fast as possible.

[See Appendix C *for some examples of how the origin server was configured.]*

Images from Mars Surface

All images are stored in separate directory structures, e.g.:

```
  http://www.marspolarlander.com/live/sol0/ssi/
```

Many file systems exhibit performance difficulties with large numbers of files in a single directory. It is, therefore, worthwhile to create a set of directories that helps improve performance and also ease maintenance. The structure is based on the sol (Martian day) and the image filename is based on the time and sequence number from the sol's downlink.

```
http://www.marspolarlander.com/live/sol0/ssi/s0568211380_00000003901.jp
g
```

These images, and the URLs that refer to each of them, are unlikely ever to be modified. For this reason, expiration times far in the future are set for the images, enabling them to be stored in caches and accelerators for very long times and served rapidly to end users.

There are a variable number of downlinks from Mars each sol, and images are, therefore, received in (essentially) random batches. For each image, the following files are created:

- A GIF or JPEG image.

- An HTML file “embedding” the image with an associated description.
- A “slideshow” HTML file that provides links to the previous and next images.

The slideshow pages are assigned very long expiration times, except for the most recent. This page is dynamically assigned very short expiration times while during a downlink period and waiting for the new image to be completed. When the current downlink period is completed the expiration time is automatically increased.

Archives

Image and Product archives will be created at the same time as the “live” pages are created. Index pages of these archives are generated also. They should be available at:

```
  http://www.marspolarlander.com/live/sol0/ssi/index.html
```

Example of image HTML page in Archive:

```
http://www.marspolarlander.com/live/sol0/ssi/s0568211380_0000000390l.ht
ml
```

The products will follow the same design, e.g.:

```
  http://www.marspolarlander.com/products/sol0/ssi/mosaic1.html
```

Latest Image Generation and Distribution

The following describes how the images are to be generated and distributed:

- Data (comprising an image) is received from the Mars Polar Lander by the Deep Space Network and sent to UCLA.
- The image is converted to formats suitable for Internet presentation by the SSI science team and stored on an Operations network computer.
- An image processing program is triggered on *live.mars.ucla.edu* upon the arrival of new data and an HTML page is created and stored on this server.
- Each of the accelerators is instructed to preload the image.
- A short time thereafter (5–10 seconds) the HTML page showing the latest images will “embed” the new image, when requested by users.
  The expiration time generated for the last page will refresh every 20 seconds during downlink. The other images for the current downlink that arrive before the “atest” will be contained on pages with long expiration times, as these will not be changed.

This design means that by the time users have retrieved the latest HTML page, the images are already in the caches, ready to be retrieved quickly.

Recording Access—with Caching

Access to the Web site is measured by forcing revalidation of a particular embedded image on the Web pages to be counted (only a small number of Web pages are actually counted).

So that traffic to the origin server is minimized, the revalidations are “terminated” at the accelerators. This is done by sending *Cache-Control* headers to the accelerators than are different from those sent to clients and other caches.

Appendices

A. Network Details

Network Configuration

The following table describes the network parameters at each accelerator location:

Location	Default Gateway	Subnet Mask	Broadcast Address	Available Range
Reston, VA	204.71.168.1	255.255.255.128	204.71.168.127	204.71.168.[2-3]
New York, NY	204.71.168.129	255.255.255.128	204.71.168.255	204.71.168.[130-131]
Chicago, IL	204.71.169.1	255.255.255.128	204.71.169.127	204.71.169.[2-3]
San Francisco, CA	204.71.169.129	255.255.255.128	204.71.169.255	204.71.169.[130-131]

DNS Configuration

The DNS is configured as follows:

- DNS A queries for *www.marspolarlander.com* resolve to one of the accelerator IP addresses
  (as determined by the network proximity algorithms).
- DNS PTR queries for **any** of the accelerator IP addresses resolve to *www.marspolarlander.com*.
- DNS MX queries for *marspolarlander.com* resolve to mail hosts at *ucla.edu*.

3DNS Controllers manage the *www.marspolarlander.com* domain; one is co-located with each NetCache:

Location	3DNS FQDN	3DNS IP Address
Reston, VA	lbrst.cw.net	204.71.168.3
New York, NY	lbnyd.cw.net	204.71.168.131
Chicago, IL	lbchd.cw.net	204.71.169.3
San Francisco, CA	lbsfd.cw.net	204.71.169.131

It is desirable, for administrative reasons, to be able to refer to the accelerators individually, so there are entries for each accelerator in the *marspolarlander.com* domain:

Location	NetCache FQDN	NetCache IP Address

Reston, VA	accel-rst.marspolarlander.com	204.71.168.2
New York, NY	accel-nyd.marspolarlander.com	204.71.168.130
Chicago, IL	accel-chd.marspolarlander.com	204.71.169.2
San Francisco, CA	accel-sfd.marspolarlander.com	204.71.169.130

There are very few instances when DNS resolving is required by the accelerators:

- Client accesses are logged by IP address; a reverse lookup for their FQDN is not necessary.
- Server access is always to a single origin site; the DNS query for this IP address can be cached for a very long time.

The following two DNS servers are used, if necessary:

FQDN	IP Address
pdns1.isc.cw.net	208.134.245.2
pdns3.isc.cw.net	208.134.245.10

B. NetCache Validation Details

NetCache generates a Time-To-Live (TTL) for all objects that are cached. Objects will not be revalidated with the origin server until the TTL expires.

Explicit Expiration Time?	Last Modified Time?	Cache?	If-Modified-Since Time	Time To Live
Yes	Yes	Yes	`last_verify`	`max_age` or `(Expires - last_verify)`
Yes	No	Yes	`last_verify`	`max_age` or `(Expires - last_verify)`
No	Yes	Yes	`last_verify`	`(now - last-mod)/2`
No	No	No	-	-
No	No	Yes *See notes below*	`last_verify`	*Default value in UI*

Note:

- *last_verify* is the time the object was last validated with the origin server.
  This is usually the time-stamp in the *Date* response header.
- *If-Modified-Since Time* is the time used in the *If-Modified-Since* request header of a conditional GET.

- The *Cache-Control: public* response header will force a page to be cached, regardless of other conditions.
- The Default TTL can be configured by the enduser.
- It is possible to override the TTL generated by NetCache and set specific TTLs for any URLs (matching by regular expression).
- The configurable *Refresh Rate* parameter allows the setting of a maximum age for all objects that are cached. When this age is reached, revalidation **will** occur.

C. Origin Server Configuration Examples

The *Apache* Web server was compiled with the *mod_expires* and *mod_headers* modules, which allowed explicit expiration times to be set for URLs in a simple and flexible manner.

The following statements were used at the server configuration level to set a default expiration time of 1 month for all URLs:

```
ExpiresActive On
ExpiresDefault "now plus 1 month"
```

(Note that the *mod_expires* module adds both *Expires:* and *Cache-Control: max-age* headers.)

To allow finer-grained control over particular URLs, *.htaccess* files were placed in appropriate locations.

(Note that parsing *.htaccess* files can be CPU intensive and, where possible, these configuration statements should be placed in the main server configuration file.)

So that the main page with news updates could be updated periodically, a low expiration time was set using the following *.htaccess* file:

```
<FilesMatch "index.html">
  ExpiresDefault "modification plus 1 hour"
</FilesMatch>
```

To set the desired headers on the embedded image used for tracking, the following *.htaccess* file was used:

```
<FilesMatch "tracker.gif">
  ExpiresActive Off
  Header set X-Accel-Cache-Control max-age=86000
  Header set Cache-Control "must-revalidate, public, max-age=0"
</FilesMatch>
```

For much of the content that was generated by automated scripts (mostly written in Perl), *.htaccess* files were generated that overrode the default behavior as appropriate.

PHP was used on a number of pages for a variety of reasons (e.g., assembling the mosaic info). Functions were used that allowed appropriate headers to be set, e.g.:

```
header("Cache-Control: max-age=3600");
```

Note that NetCache versions 4.1 and below do not update HTTP response headers that are received with a 304 *(Not-Modified)* response (a bug). This means that if a file is not modified but the metadata is (e.g., increasing the expiration time by modification of the appropriate *.htaccess* file), then it is necessary to ensure that the file *appears* modified to the Web server so that NetCache will update the headers as appropriate (a simple method to achieve this is to use the UNIX *touch* command).

D. Results and Conclusions

Conclusion This report describes the architecture of the site and the issues that it was designed to overcome:

- The traffic from the Internet was to be directed *away* from the UCLA network as there was not sufficient capacity in the Internet link to handle an event of this magnitude.
- The load on the origin servers would have been very high and possibly unmanageable.
- Response Time is always a problem for end users, and more bandwidth does not help as data still has to travel the same distance. Bringing content closer to end users is the *only* way to solve this.

These issues were solved by:

- Leveraging the existing Internet infrastructure (caches are now deployed worldwide) and designing the site to be very cacheable. This brings the content closer to end users, reducing latency, and also reduces the amount of requests that must be satisfied by the Web site.
- Using distributed Web accelerators, thus redirecting traffic away from the origin servers.

Initial Difficulties

There was a miscommunication in publicizing the URL for the Web site and the press consequently promoted an internal UCLA URL as the official address. This caused tens of thousands of Internet users to attempt to connect directly to the UCLA network on the day the spacecraft arrived on Mars (Friday, 3 December 1999).

Because of this, UCLA essentially experienced an enormous (but unintended) *Denial of Service* attack.

Meanwhile, the NetCache accelerators were receiving many requests for the correct Web site address and were handling the load without any difficulties.

However, as you will understand from the report, NetCache periodically needs to revalidate cached pages with the origin server. For the start of this day, most of the URLs had short expiration times as last-minute changes were being made. When the URL began to expire on the accelerators and required revalidation, the origin server was incapable of replying to the simple requests and hence the accelerators were unable to return content to the clients.

Mars Polar Lander operations were to be conducted from UCLA and all network elements were under configuration lockdown, which made it difficult to respond to the attack rapidly. The scale of the assault was such that it started to impact negatively the capability of the network to handle operations and, ultimately, because of this, it was possible to modify the routers to block all Web requests entering the network unless they came from the accelerators.

Results

After the initial difficulties was corrected, the Web site worked flawlessly. Unfortunately, because the spacecraft was lost, the site was not as popular as was hoped, serving about 21 million requests through the first 3 days—less than 10% of the anticipated load.

However, although there were some initial difficulties, from a technological point of view the implementation was quite a success. The following concepts were all validated:

- Caches are now deployed worldwide, and designing Web sites to take advantage of this gives huge benefits, allowing them to scale enormously.
- Distributed Web acceleration further enhances this by moving any loads out onto the network and away from the origin server.
- It is possible to manage content in a single location and allow the "network infrastructure" to distribute it—there is no need to have the administrative overhead of maintaining multiple origin servers or mirror sites.

Because the sites were designed explicitly for cacheability, each of the accelerators showed between 98% and 99% object **and** byte hit rates. Figure 6 shows the ratio of client to server connections was very high (i.e., only 1 or 2 connections to the origin server were being made while hundreds of client requests were being handled).

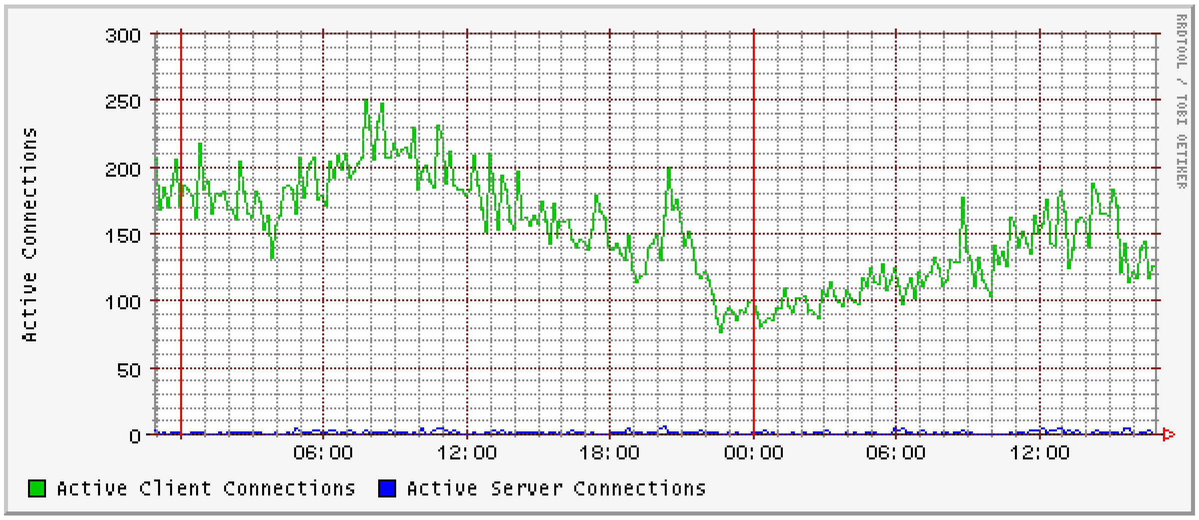

Figure 6: Active client and server connections to/from one of the accelerators (Saturday/Sunday, 4/5 December 1999). Note that this represents a relatively low load.

The site also demonstrated that the design used for recording access works very well. A small image was embedded on some pages and specialized headers were applied to it. Clients and caches were forced to revalidate this object and these revalidations were *terminated* at the accelerators. Without making a page uncacheable, or overloading an origin server, this is one of the few ways that accurate recording can be done, and we have shown that it can be made to work reliably!

Glossary

Accelerator
    A cache dedicated to a select number of Web sites.
Cache

A device capable of storing frequently accessed Web content closer to the end users, resulting in reduced bandwidth consumption and response-time improvements.

Client

A program that establishes connections for the purposes of sending requests.

Origin Server

A HTTP server where content originates.

Server

A program that accepts connections in order to service requests by sending back responses. Any given program is capable of being both a client and a server (e.g., a cache).

Web site

The combination of origin server(s) and accelerator(s) that responds to client requests.

References

[HOSTING]
Network Appliance - Mars Polar Lander: Powered by NetCache.

[DNS1998]
Paul Albitz & Cricket Liu, *DNS and BIND, 3rd Edition*, September 1998.

[MNOT1999]
Mark Nottingham, *Caching Tutorial for Web Authors and Webmasters*, June 1999.

[FNDRY]
Foundry Networks, Application Note: Global Server Load Balancing

[RFC-1738]
T. Berners-Lee, L. Masinter, M. McCahill, *Uniform Resource Locators (URL)*, December 1994.

[RFC-2616]
R. Fielding et. al., *Hypertext Transfer Protocol—HTTP/1.1*, June 1999.

[APACHE]
The Apache Software Foundation, *The Apache Server Project.*

[PHP]
PHP: Hypertext Preprocessor.

[PERL]
The Perl Programming Language.

[MYSQL]
The MySQL Server.

NetApp®

Network Appliance, Inc.
495 East Java Drive
Sunnyvale, CA 94089
www.netapp.com