On Defining a Role for Demand−Driven Surrogate Origin Servers

Mark Nottingham <mnot@akamai.com>
Akamai Technologies San Mateo, CA, USA

Abstract

This paper examines the common uses and characteristics of demand−driven surrogate origin servers (also known as Reverse Proxies and HTTP Accelerators), attempts to motivate the definition of a distinct role for them, suggests items for consideration during the development of such a framework, and identifies issues associated with current implementations.

The author's primary goal is to encourage discussion and further research of these points.

1 Introduction

A surrogate origin server (also known as a reverse proxy or HTTP accelerator) is a device that serves requests on behalf of an origin server (known as its master origin server)[1].

Demand−driven surrogate origin servers are populated by the traffic flowing through them; when a client requests an object that is not resident, the surrogate will fetch it from its master origin server. Note that a demand−driven surrogate is distinct from a replica, which does not necessarily use the HTTP to communicate with the master, and is generally not populated by traffic flowing through it; instead, content is pushed out to replicas as it is published.

1.1 Previous Work

Although demand−driven surrogates have been used for some time, most previous literature has focused on performance issues[16], benefits of their use[14], and issues such as methods of directing requests to them[15].

A surrogates birds−of−a−feather meeting was organized by Ted Hardie at the 4th International Web Caching Workshop in 1999[11], which resulted in the establishment of a surrogate−specific mailing list[12]. The IETF Web Replication Working Group[4] has a taxonomy document[1] in process that establishes terminology such as 'surrogate' and 'demand−driven'.

1.2 Current Practice

Currently, many proxy vendors (such as Inktomi[5], Network Appliance[6], Novell[8] and NLANR[9]) allow their products to be used in "HTTP Accelerator" or "Reverse Proxy" mode, where the device is configured to cache and forward traffic to a particular Web server. So−called "Content Delivery Networks" such as Akamai[10] and Digital Island[7] also leverage proxy software in a similar manner, to distribute content so that it is close to users.

Generally, these devices attempt to conform to the requirements for a proxy, as specified in HTTP/1.1[2]. To operate as a surrogate, they "reverse" themselves, so that they appear as an origin server to clients, accepting only traffic for their master origin server.

In normal operation, demand−driven surrogate origin servers are deployed and maintained by (or on behalf of) the publisher of a Web site, rather than directly or indirectly for end users, as a proxy would be. This is done for a number of reasons, including:

- Reduction of load on the master origin server
- Reduction and flattening of network traffic to the master origin server
- Distribution of objects to the 'edge' of the network, in order to improve user−perceived latency
- Introduction of content transformation or other value−added services
- Increased security, by forcing clients to communicate through an intermediate

Surrogate deployments have been observed to vary in several ways, including:

- Proximity − surrogates may be deployed close to the master origin server to reduce load on it, or near end users to reduce network traffic and improve perceived latency.
- Selection of surrogate objects − entire Web sites may be routed through surrogates, or a subset of a site's objects may be nominated for routing through them, depending on the effect desired, and the nature of the surrogate.
- Number of surrogates − surrogates may be deployed in any number, from one to thousands.
- Number of master origin servers − a surrogate may be configured to accept traffic for multiple master origin servers, identifying the appropriate master through examination of request characteristics (such as Host header or Request−URI).

- Directing traffic to the surrogate − any number of mechanisms may be used to distribute requests between multiple surrogates, including DNS or BGP manipulation, Layer 4, Layer 7 or "Content Aware" switching[13], and reference modification.
- Request and response modification − surrogates may perform any number of actions which may transform either the request (particularly the Request−URI), the response (e.g., transcoding), or both.

2 Motivating a Surrogate Role

Proxies have a well−defined role in the HTTP;

> An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them on, with possible translation, to other servers. A proxy MUST implement both the client and server requirements of this specification. A transparent proxy is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. A non−transparent proxy is a proxy that modifies the request or response in order to provide some added service to the user agent, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering. Except where either transparent or non−transparent behavior is explicitly stated, the HTTP proxy requirements apply to both types of proxies.

Demand−driven surrogates operate in a similar fashion, in that they implement both client and server requirements, and may also implement a cache. These similarities are the most likely cause of the widespread use of proxy software to effect a surrogate.

However, surrogates are used in a very different context; they appear to downstream clients as an origin server; the surrogate's address is part of the location identifier[3] for a resource it serves. Because of this, a surrogate can be more accurately classified as a gateway;

> A server which acts as an intermediary for some other server. Unlike a proxy, a gateway receives requests as if it were the origin server for the requested resource; the requesting client may not be aware that it is communicating with a gateway.

A protocol−correct definition of a surrogate would thus be a gateway which uses HTTP on its upstream connection and which may implement a cache. While this satisfies the need to define the basic requirement for such a device, it does not address the expectations of current users, or the opportunities that such devices offer.

One underlying reason for this is the general nature of requirements for gateways in the HTTP; in most cases, the specification makes little distinction between the requirements for a proxy and a gateway. The gateway requirements are intended to be applied to any device which acts on behalf of another server, without regard to the communication protocol between them.

This can cause some conflict when considering cache implementation. A client cache acts on behalf of a client, either in a user−agent a proxy. Caches are encouraged to exhibit semantic transparency;

> A cache behaves in a semantically transparent manner, with respect to a particular response, when its use affects neither the requesting client nor the origin server, except to improve performance. When a cache is semantically transparent, the client receives exactly the same response (except for hop−by−hop headers) that it would have received had its request been handled directly by the origin server.

Semantic transparency is a necessary quality for a client cache, because the proxy cannot represent the origin server faithfully; they do not have a trust relationship. Surrogates, however, do have such a relationship by nature, which allows them to act on a server's behalf in what may be a semantically non−transparent fashion.

Notice that a non−transparent surrogate does different things than a non−transparent proxy, which reduces transparency "in order to provide some added service to the user agent"; more often than not, a surrogate reduces transparency in order to add service on the origin server's behalf.

A related term in the HTTP, first−hand, also deserves reconsideration in the light of surrogacy;

> A response is first−hand if it comes directly and without unnecessary delay from the origin server, perhaps via one or more proxies. A response is also first−hand if its validity has just been checked directly with the origin server.

Because surrogates operate in close concert with origin servers, it would be advantageous if their responses were considered first−hand as well. As we will see, considering surrogate's responses first−hand makes them much more useful, but also increases the need for standard methods of controlling them, and careful consideration of the impact of such measures.

Finally, there are several second−order requirements in the HTTP for gateways and caches which arguably should not apply to surrogates because of their nature. A separate role for them would enable clarification of these issues. For example, clients are given mechanisms (such as Cache−Control and Pragma request headers) which allow them to communicate how the cache should handle requests on their behalf. In a surrogate model, these behaviors may be counterproductive.

Considering the opportunities offered by surrogates, potential pitfalls in their implementation, and the state of current implementations, it would be advantageous to define a distinct role for surrogates. Such a definition could take form as a group of extensions, recommendations and further requirements for a gateway which uses HTTP for upstream connections and implements a cache, but should be distinct from these more general roles, to reduce confusion.

3 Considerations for a Surrogate Role Definition

Defining a framework for demand−driven surrogate origin servers necessitates consideration of many factors. This section identifies relevant issues for further discussion and research.

3.1 Surrogate Configuration

At the most basic level, a surrogate needs to be configured with the identity of an origin server (or servers) to forward traffic to. However, it is often desirable to configure surrogates with other information, including:

- Encryption or authentication information required by the master origin server
- Default object handling information, including coherence
- Specific object handling information
- Access control lists
- URI rewriting
- Other special instructions to the surrogate

In general, configuration of surrogates is needed to allow them to represent the origin server accurately, and to take full advantage of the opportunities they afford. For configuration to take place, there must be a mechanism that specifies how it is to be applied to request and response messages that the surrogate handles.

Possible configuration mechanisms include:

- In−Request − within the request−URI itself, or using Request HTTP headers. For instance, the master origin server identity may be contained in the Request−URI submitted to the surrogate, but stripped from it before it is forwarded to the master, or it may be derived from the Host request header.
- In−Response − some configuration, especially object handling information, may be communicated in response HTTP headers from the master origin server.
- Manual − configuration through a console, terminal or other local interface.
- Out−Of−Band − configuration may be possible through a remote mechanism which, for example, may resemble an RPC interface, or through a separate file which dictates how to apply configuration. This is particularly effective for configuration of large numbers of surrogates.

Standardizing both configuration mechanisms and common configuration directives is an interesting topic for future work. In−Response mechanisms easily lend themselves to standardization. For example, a "Surrogate−Control" response header, which could be used to communicate configuration information, including freshness control, in a manner similar to Cache−Control response headers. Out−Of−Band configuration has similarly interesting possibilities. No matter what the configuration mechanism, interactions between configuration directives and HTTP mechanisms should be carefully considered.

3.2 A New Consistency Model

Cache consistency is important to surrogates because of their scope, and content publishers' expectations. If a client−based cache is incoherent, only those clients using it will be affected; if a surrogate cache is incoherent, potentially all clients for a service will be.

The basic tools of caching (freshness and validation) are still useful to a surrogate. However, they are used in a different context; surrogates typically serve a smaller, more defined footprint of objects, and their close relationship with origin servers allows them to cache content differently than a downstream cache might. For example, a content publisher who needs to record every access to an object will mark it uncacheable. Since surrogate logs are usually available to the publisher, it would be useful to have a way to allow the surrogate to cache the object, while still marking it uncacheable downstream.

One way to effect this is to separate the coherence mechanisms between origin servers and surrogates from normal cache coherence. This allows surrogate caches to operate efficiently, while different coherence information is available for non−surrogate caches downstream. It also introduces the possibility of using enhanced coherence mechanisms between the origin server and surrogate.

One possible implementation is the use of surrogate−specific response headers which complement Cache−Control headers. For example, a response containing:

```
Surrogate−Control: max−age:600
Cache−Control: max−age:300
```

would instruct the surrogate to assign a 600 second freshness to the object, while downstream caches would keep it fresh for 300.

It is also advantageous to give content publishers the ability to manipulate the consistency of objects already in cache (e.g., marking them stale, or purging them from cache). This is possible because the identity of surrogates are typically known, and their numbers limited, unlike caches in proxies or user agents.

Such a consistency model gives content publishers the ability to directly and precisely control surrogate coherence. Under such levels of control, it is possible to consider responses from a surrogate (including cached responses) as authoritative.

3.3 Redefining the Connection

End−to−end headers in the HTTP are terminated at the client and origin server. Because a surrogate is not semantically transparent, it is necessary to evaluate end−to−end headers to ensure that their intent in the protocol is preserved.

It is useful to consider surrogates as the endpoint for some end−to−end headers for both upstream servers and downstream connections, because of the different context they operate in. In particular, a surrogate role may require rethinking the concepts surrounding identity. For example, surrogates may represent themselves in the Server header sent to clients, rather than forwarding it from the master origin server. The Host request header should also be modified to reflect the identity of the master origin server, rather than the surrogate.

Because the surrogate is addressed in the URI used to contact it, SSL connections cannot be tunneled through it to the master, as would happen with a proxy. Instead, separate certificates must be maintained for surrogates and master origin servers.

Separating the connection at the surrogate also allows different transports and features to be used on the upstream and downstream connections. For instance, it may be desirable to fetch object from the master origin server while protected by SSL (for authentication and security of the surrogate/master relationship), while doing so is not necessary for connections with clients.

Surrogates can also be used to add protocol features not found on the master origin server, whether they are hop−by−hop or end−to−end. For example, surrogates may be used to add Transfer−Encoding capabilities, add synthetic validators or other cacheability−enhancing information, and transcode content depending on client capabilities or other external criteria.

3.4 Relationships with Other Intermediates

Generally, a surrogate has similar relationships to that of a proxy; it acts as a client to upstream servers, and a server to downstream clients (although the nature of this interface is different, as previously noted).

However, a surrogate's relationship with its master origin server requires special consideration. Because semantic transparency is not necessarily maintained by surrogates, there may be complications if they are deployed in a hierarchy as children or parents of other surrogates.

3.5 Request−URI Modification

Some implementations may modify (rewrite) the Request−URI, in order to extract configuration information, or to enforce their configuration in respect to master origin server load balancing, transcoding, or other special considerations.

This introduces the need for a terminology that identifies the Request−URI before and after it has been modified. Suggested terms are:

- Published−Request−URI − the Request−URI before modification; that published for referencing by clients.
- Origin−Request−URI − the modified Request−URI, used to locate the requested object on the master origin server.

4 Observed Issues in Current Implementations

Several issues have been observed when proxy/cache software has been used as the basis for a surrogate. Many stem from the issues raised above, and should be addressed by a surrogate framework specification.

4.1 Date Headers and Age Calculation

In HTTP/1.1[2] The Date response header is required to reflect the time that an object is generated on its origin server. Caches are directed not to modify an object's Date header if one is present. Additionally, they are required to include an Age header;

> The age of a response is the time since it was sent by, or successfully validated with, the origin server.

This aging mechanism is designed to assure that the response is kept coherent with the origin server, no matter how many caches it has traveled through.

If coherence between the surrogate and master is separate from other cache coherence, surrogates may wish to represent themselves as authoritative for those objects, by removing the Age header and updating the Date to reflect the current time. Doing so allows the standard HTTP coherence mechanisms to operate as intended.

Failing to update the Date header and remove the age can affect downstream cacheability, so that it is not predictable.

4.1.1 Misinterpretation of Cache−Control: max−age

HTTP/1.1 states that the age of an object in a cache can be calculated as:

```
apparent_age = max(0, response_time − date_value)
resident_time = now − response_time
current_age = apparent_age + resident_time
```

(this is somewhat simplified; see the specification[2] for the full algorithm).

If date_value reflects the time that an object first entered a surrogate, rather than the time that the surrogate serves the object to its client, apparent_age may be unnaturally large, depending on how long is has been resident on the surrogate.

This has the effect of artificially inflating the value of current_age, causing the object to become prematurely stale.

For example, imagine an object that enters the surrogate at an imaginary time, 1200, is accessed by a downstream cache at time 1500. If the object has a max−age of 600, and the downstream cache needs to make a decision about its freshness at time 2000, the calculation is:

```
apparent_age = max(0, 1500 − 1200) = 300
resident_time = 2000 − 1500 = 500
current_age = 300 + 500 = 800
```

In this case, the calculated age (800) exceeds the specified max−age (600) by 200, causing it to be considered stale.

This is undesirable, because the surrogate holds authoritative instances of objects for the master origin server. If the Date header is current when objects are served from the surrogate, the calculation becomes:

```
apparent_age = max(0, 1500 − 1500) = 0
resident_time = 2000 − 1500 = 500
current_age = 0 + 500 = 500
```

In this case, the calculated age (500) is within the freshness lifetime of the object (600).

4.1.2 Miscalculation of heuristic freshness

In a similar manner, objects without explicit freshness information may be adversely affected by a surrogate which does not update Date headers.

The HTTP allows caches to apply a heuristic freshness mechanism to such objects. The most common is expressed[17]:

```
cache_age = now − date_value
lm_age = date_value − lm_value
lm_factor = cache_age / lm_age
```

where lm_factor is compared to some configured value to determine the freshness of the object.

In this case, propagating the original Date header has greater potential to skew downstream object freshness. Imagining that the object was last modified at 1000;

```
cache_age = 2000 − 1200 = 800
lm_age = 1200 − 1000 = 200
lm_factor = 800 / 200 = 4
```

If the Date header is updated, however,

```
cache_age = 2000 − 1500 = 500
lm_age = 1500 − 1000 = 500
lm_factor = 500 / 500 = 1
```

The effects of using a cached Date header upon heuristic freshness are unpredictable, as they are highly dependent on the time that the object first entered the cache.

4.1.3 Interpretation of Expires as a delta, rather than absolute time

Expires headers allow content publishers to specify a time when the object will be considered stale. HTTP/1.1 calculates the freshness lifetime of an object with an Expires header as:

```
freshness_lifetime = expires_value − date_value
```

Because of this, if a surrogate does not update an Expires header, downstream caches will treat the object as fresh for the period of time between the Expires and Date headers, rather than expiring it at the specified time.

If a Cache−Control: max−age response directive is set, origin servers may set a complimentary Expires: value, to duplicate the intended freshness delta for HTTP/1.0 clients. To accommodate this, surrogates that update Date headers should recalculate the Expires header to match the delta communicated in Cache−Control: max−age, but only if both are present in a response, and are equivalent.

It has also been noted that some older Web servers set an Expires header based on a delta from the Date, without setting a Cache−Control: max−age header. This is problematic, as it is difficult to distinguish these responses from those which wish to expire content at an absolute date.

Cacheable responses which include a Set−Cookie header with its own specified expiry will be similarly affected.

4.2 Proxy−Specific Directives

There are several directives and mechanisms in HTTP/1.1 that have special meaning to proxies;

- CONNECT method − is reserved for establishment of a tunnel through a proxy, and is not applicable to surrogates.
- TRACE method − allows remote loop−back of the request. Surrogates that allow TRACE through them may inadvertently expose the master origin server and communications between it and the surrogate.
- Cache−Control request headers − allow clients to specify how a proxy should handle the request. For surrogates, this may not be advisable in all situations, and is not necessary because of the tight binding between the surrogate and its master.
- Pragma request headers − similar to Cache−Control request headers, Pragma allows clients to control request handling by proxies.
- Proxy−Authenticate and Proxy−Authorization − these headers (response and request, respectively) have little meaning to a surrogate, as they are designed to limit use of a shared proxy/cache.

4.3 Client Authentication

HTTP client authentication (through use of WWW−Authenticate and Authorization headers, as well as 401 status codes) may be effectively propagated through a surrogate, by considering such objects public, and enforcing validation upon them, to assure proper authentication.

Delivery may be further optimized by caching the authentication state of clients on the surrogate, to allow it to serve objects from cache without revalidation on the master origin server.

4.4 Content Negotiation

Surrogates have a unique opportunity to serve object variants based on request attributes. This can be accomplished by creating variants based on configuration information, or by rewriting requests to the origin server based on request attributes. This effectively moves the variant mapping from the origin server to the surrogate.

4.5 Redirect Resolution

Proxies are required to forward redirects (301, 302 and 307 status codes), just as any other status code from the origin server.

Because redirects are used for a variety of purposes, including relocation of resources, primitive content negotiation, and other specialized applications, proxies cannot derive the proper behavior to exhibit, and must forward the redirect to the client, despite the strong possibility that the result is cacheable.

Surrogates, on the other hand, are not necessarily bound by this requirement. Configuration can be used to identify certain classes of redirection so that only the resulting object is served to the client, in turn offering a substantial gain in efficiency.

It may be productive for future work to define appropriate configuration directives and associated behaviors to more effectively describe redirect handling in surrogates.

4.6 Content Integrity and Security

Because a surrogate serves objects authoritatively, content integrity may be perceived as more critical than in a proxy/cache. As a result, surrogates should support HTTP integrity mechanisms such as the Content−MD5 response header, and may need to develop external mechanisms for assuring integrity. Proper behavior when an integrity error is detected also needs to be considered.

In some cases, a third−party proxy/cache deployed between a surrogate and its master origin server may distort the relationship between them. Surrogates may need to take measures such as appending Cache−Control and Pragma headers when making requests to the master origin server, in order to assure that the correct response is retrieved.

For some applications, it may be desirable to force clients to use a surrogate, and disallow direct access to the master origin server. This may be accomplished by any of a number of mechanisms, including

- Client−side SSL certificates
- HTTP authentication into a surrogate−specific authentication realm
- A surrogate−specific Cookie as an identifier

4.7 Logging

Proxy−specific log formats may not be appropriate for use by a surrogate.

Logs on Web servers are used for many purposes, and content publishers often design their sites to maximize available information in origin server logs by modifying the cacheability of resources and logging identifiers such as cookies, referrers and user agents.

Surrogates should be capable of logging such information, in a manner compatible with common origin server logs, to enable such functions to be moved to the surrogate, rather than forcing requests to be made back to the origin server for these reasons.

Another issue sometimes faced is what to log as the Request−URI; if a surrogate modifies it, there may be some situations where it is desirable to log the original Request−URI, and others where the Request−URI to the master origin server is needed.

5 Conclusions and Future Work

There are many issues, both obvious and subtle, caused by the use of proxy software as a surrogate origin server. These are amplified by the lack of a framework or specifications for them. As surrogates lend themselves to being deployed for high−traffic sites, the potential for problems with them and opportunities for extending their functionality and efficiency are considerable.

This paper establishes talking points that will help start discussion of the issues identified, in order to bring these benefits about and help avoid problems identified. Further work may include establishment of a role and more specific research into the behaviors of surrogates.

6 Acknowledgments

The author would like to thank John Dilley, Peter Danzig, Ted Hardie, Edith Cohen and Roy Fielding for their contributions and comments.

7 References

(1) Cooper, I., Melve, I. and G. Tomlinson, "Internet Web Replication and Caching Taxonomy", draft−ietf−wrec−taxonomy− 03, March 2000.
(2) Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P. and T. Berners−Lee, "Hypertext Transfer Protocol − HTTP/1.1", RFC 2616, June 1999.
(3) Berners−Lee, T., Fielding, R.T. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.
(4) http://www.wrec.org/
(5) http://www.inktomi.com/
(6) http://www.netcache.com/
(7) http://www.digisle.com/
(8) http://www.novell.com/
(9) http://www.squid−cache.org/
(10)http://www.akamai.com/
(11)http://workshop.ircache.net/BOFs/bof2.html
(12)surrogates@equinix.com (subscribe: surrogates−request@equinix.com)
(13)http://www.arrowpoint.com/switch/white_papers/cache_switching.html
(14)http://www.novell.com/bordermanager/accel.html
(15)Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. "Web Caching with Consistent Hashing", Proceedings of the 8th International World Wide Web Conference, May 1999. http://www8.org/w8−papers/2a−webserver/caching/paper2.html
(16)Levy−Abegnoli, E., Iyengar, A., Song, J., and Dias, D. "Design and Performance of a Web Server Accelerator", Proceedings of the IEEE Infocom ’99 Conference, March 1999. http://www.research.ibm.com/people/i/iyengar/infocom1.ps
(17)D. Wessels et. al., Release Notes for Version 1.1 of the Squid Cache. July 1997.

8 Vitae

Mark Nottingham is a Senior Developer at Akamai Technologies. There, he helps specify the behavior Akamai servers should exhibit when interacting with the HTTP, as well as defining an interface for Akamai server configuration.