New Mod_proxy Features and Uses for httpd 2.0

Graham Leggett
minfrin@apache.org
Chuck Murcko
chuck@apache.org
ApacheCon 2002
TU09

Abstract

This presentation will discuss the httpd-2.0 mod_proxy module and its protocol-specific submodules. Mod_proxy is introduced with a discussion of its uses, as well as a brief history of its development. Mod_proxy can be used to solve a number of common problems deploying and scaling web server farms, and a number of possible solutions are illustrated. The architecture of the v2.0 mod_proxy is introduced and explained, and changes from v1.3 are described. Potential enhancements are discussed, along with ideas for their implementation.

Introduction to mod_proxy

What Does It Do?

In a typical web server, an URL would be served based on the contents of a static file on a disk, or the result of execution of a program (CGI) or server side script language (PHP/JSP). Mod_proxy allows the web server to serve URLs based on the content mapped from other URLs. This allows web server and web application systems to be built out of multiple tiers, solving a number of common scaling and maintenance problems involved while managing a web server farm.

Mod_proxy also allows Apache to behave as an HTTP proxy as described by RFC2616.

History of mod_proxy

David Robinson originally wrote mod_proxy in 1995, shortly after the adoption of Rob Thau's Shambhala code base as httpd 0.8. Ben Laurie rewrote the original caching code, and Chuck Murcko took over maintenance of the module when David went emeritus from the (then) Apache Group. The main goal in the 1996-1998 timeframe was to move mod_proxy from experimental to production state. During this period, and through httpd releases 1.0-1.2, many contributed fixes and features (among them the original CONNECT method code) and passive mode FTP were added. Mod_proxy was split into a number of submodules (it had been a single monolithic code block) based on application layer protocols, and converted to a UNIX archive. Early address/namespace filtering was also added at this time. Dean Gaudet originally suggested adding an API to mod_proxy for easier addition of protocols.

Chuck went emeritus from the Apache Group for 1-1/2 years, and maintenance was taken over during the httpd 1.3 release cycle by a group of people, including Martin Kraemer, Graham Leggett, and Dirk-Willem van Gulik. Graham Leggett and Ryan Bloom have done extensive work since 2000 redesigning mod_proxy, porting it forward to the evolving httpd 2.x framework, and adding HTTP 1.1 capability to httpd 1.3.

Applying mod_proxy

Proxy can be used in two main scenarios, a **forward proxy** or a **reverse proxy**.

A forward proxy is a server that runs on or behind a bastion host in a firewall, fetching URLs on behalf of HTTP clients who are unable or not allowed to fetch the URLs directly. Although Apache can be a forward proxy, there are many other servers such as Squid dedicated to the job of forward proxy that at the moment perform better.

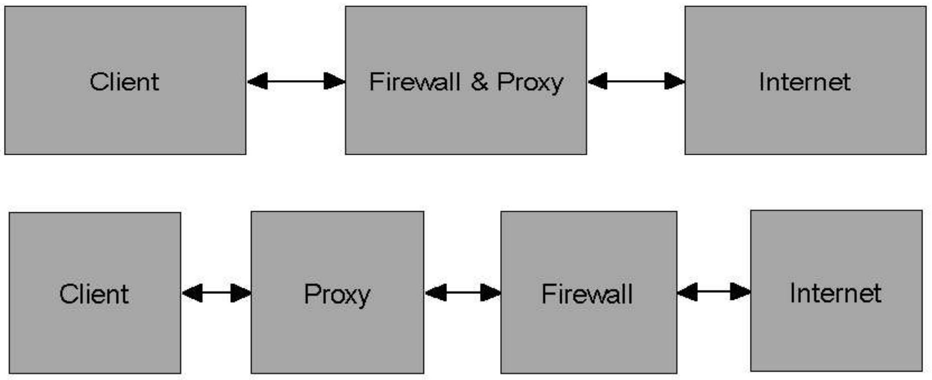

Figure 1. Types of forward proxies

A reverse proxy is a web server that is configured to tunnel requests for URLs to other back end web servers. The back end web servers are probably not directly accessible to the HTTP client that initiated the original request. A reverse proxy forms part of a multi-tiered web server design. The Apache web server does a very good job at being a reverse proxy.

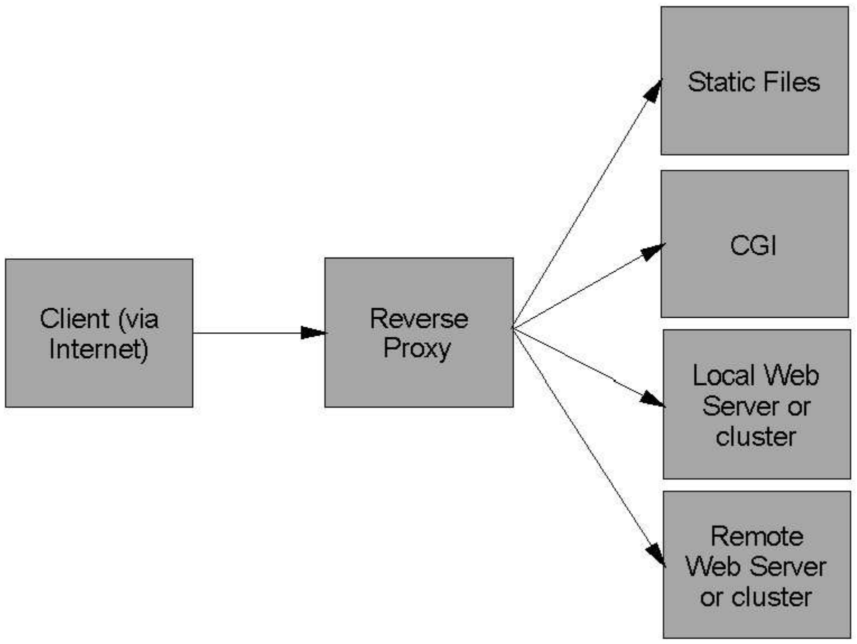

Figure 2. Multi-tier web server with reverse proxy

Using a multi-tier web server design allows you to deal with a number of common web server scenarios:

- **Security**

One of the most useful applications for a reverse proxy is for providing access to web resources, while at the same time ensuring that these resources are protected from access by the HTTP client, or by hostile networks.

Typically the **front end web server** will be attached to the internet or intranet, and will simultaneously be attached to a private backside network containing **back end web servers** to be secured. These back end web servers will not be directly accessible at all to HTTP clients. Should the back end servers be configured so that they cannot make contact with any outside nets, it becomes harder for known bugs with web server software to be exploited on the back end servers.

- **URL Management**

In a typical large organisation, there is likely to be a large number of different web server technologies that are used simultaneously. These web servers could be simple servers serving static pages, to application servers serving dynamic content or a web-based application. Each of these servers is likely to to run on different operating systems, or will require different web server or application server software to run.

Typically these different installations are made directly accessible to the internet or intranet, and HTTP clients jump from machine to machine directly by following links embedded in web pages. These links often involve high port numbers, or pure IP addresses, and are generally ugly. Maintaining such a system becomes difficult, as taking a server offline or moving a service around requires that all the links to that service to be changed, which is not always practical.

If a reverse proxy is placed in front of these different servers, it becomes possible to map all the back end server's disparate URL spaces into a single, well-managed URL space. It becomes irrelevant under what URL or on what server a particular resource is being hosted, as this resource is mapped to a well-known URL.

This has a number of advantages:

- The organisation's URL space can be centrally managed. It is no longer possible for individual server owners to "do their own thing" with the URLs they use.

- There is now no need to worry about what platform the web server needs to be on the back end. If one department requires a specific server platform different from another department, they can both be accommodated within the same URL space without a problem.

- Managing back end servers becomes significantly easier, as no longer do links need to be changed, or DNS needs to be altered to move a server. The mapping from the old server to the new becomes an issue of updating the reverse proxy configuration file and gracefully switching over.

- **SSL Acceleration**

In organisations running application servers, the web server is often tasked with both rendering web pages, and encrypting the connection between server and HTTP client. Separating these two functions into back end application servers, and front end SSL accelerators allows these two functions to be scaled separately from each other.

An Apache reverse proxy configured with mod_ssl can behave as an SSL accelerator.

- **Centralised Logging**

In a large server farm with disparate web servers, application servers and platforms, it becomes a significant problem collating log files in different formats from each server, and analysing them in a meaningful way.

Fronting these servers with a reverse proxy allows logging and statistics gathering to be performed centrally, using a common log file format.

- **Centralised Authentication/Authorisation**

Different web servers and applications handle authorisation and authentication in significantly different ways. It is often not possible to support certain types of authentication across all the different web server platforms. In addition, different URLs and domain names mean that the scope of authentication can be limited, and single sign on becomes hard to implement.

Configuring a reverse proxy to handle authentication and authorisation centrally allows a common authentication mechanism to be used with a scope across the entire URL space, independent of whatever technology is being used to render the pages on the back end.

- **Centralised Caching/Compression**

Through the use of other Apache modules such as mod_cache and mod_deflate, load can be reduced from networks and back end servers through caching of data, or the compression of data at the point where data enters and leaves the server farm.

Mod_proxy v2.0 Architecture

Mod_proxy takes the form of a parent module **mod_proxy** and a number of protocol specific modules: **proxy_http**, **proxy_ftp** and **proxy_connect**. These modules can be compiled statically into the server, or compiled dynamically and loaded at runtime.

Changes from v1.3

A number of significant changes have been made to proxy from v1.3 to v2.0:

- The cache code has been removed entirely to another module, mod_cache. Because caching is a function useful to a number of different scenarios, such as using a JSP connector, or serving static files via NFS, or caching the output of an expensive CGI process, caching is now possible for all URLs served by Apache, not just proxy URLs. In addition, this has made proxy a significantly simpler module which is now easier to maintain.

- The code is now designed to be compliant with HTTP/1.1 as described in RFC 2616. This has resulted in a number of behaviour changes from the old proxy code due to the stricter requirements of HTTP/1.1 and the handling of things like transfer encoding and content negotiation.

- Protocol handlers have been split out into individual sub modules. It is now possible to remove ftp or connect support completely from a server should this be required in configuration.

- Proxy now uses the Apache v2.0 filter mechanism in order to handle data being fetched from and served to the back end and HTTP clients respectively. This allows content returned from remote servers to be manipulated on the fly, depending on what filters have been installed.

How an HTTP request is handled

The parent proxy module registers itself as a content handler with the Apache core, and is configured to handle certain URL prefixes within the URL space using either the **ProxyPass** directive, or the **[P] flag** of mod_rewrite.

- Connections

Having determined that the request is for the proxy, the parent module passes the URL to each of the protocol specific submodules in turn to see if any of them want to deal with the request. If none of the modules want to deal with the request, or if none of the modules are loaded, proxy will return a BAD_GATEWAY.

Having determined that the request is an HTTP or HTTPS request, **proxy_http** attempts to make a connection to the remote server. If the remote server is defined as a DNS name resolving to more than one IP address, proxy will round robin between the back end servers. Should a connection to one of these IP addresses fail, it will try the next address until the IP addresses are exhausted.

- Keepalives

Proxy http supports **keepalives**. The socket is not closed at the end of a request unless specifically told to. When a subsequent request is received, a check is made to see if the new request is to the same server and port as the last request. If this is the case, the socket is reused. If not, the socket is closed and a new one opened.

Only one socket is kept open at the end of the request for the sake of keepalives. It is theoretically possible to implement a pool of sockets to the back end servers, however many of these sockets are likely to remain open for a long time, tying up resources on the back end. A pool will also result in significantly more complex code. It is unlikely to result in a significant performance win over the current behaviour.

- Send Request

Proxy http now sends the request to the back end server, consisting of the request line and the headers read from the original client request. Certain of the headers are modified according to the requirements of RFC2616 before being sent.

When the connection to the back end server is made, a new and separate filter stack is created containing the default filters capable of writing to and reading from the back end server. This results in two separate stacks being used side by side - one to handle the back end server, the second the original stack to handle the response to the HTTP client.

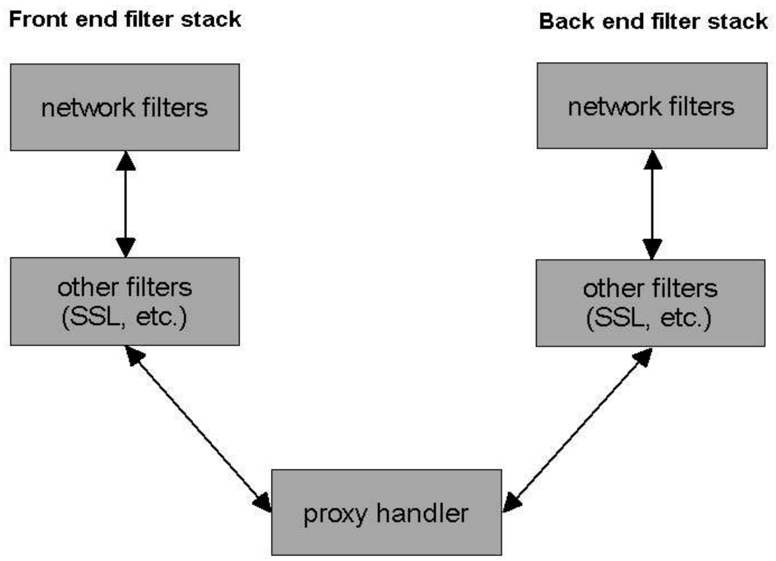

Figure 3. mod_proxy architecture

Should the HTTP client send a request body, this body is read from the client filter stack, and then written to the back end stack. Once the reading of the request is complete, proxy http now reads the response from the back end server.

As both the Apache controlled front end connection and the proxy controlled back end connection are filter stacks, it becomes possible to add the mod_ssl filters to support encrypted HTTPS requests on both the front and back end connections of the proxy.

- Read Response

Proxy now enters a loop that reads the response from the back end stack, and then passes the resultant buckets to the front end stack. When the response is complete, either because the content-length is satisfied, or chunked encoding indicates the transfer is complete, or because the connection was closed, the loop ends and the request is complete.

In order to reduce load on back end servers, a check is made to see whether the request is complete after reading from the back end server, but before writing to the front end server. If the data read was the last chunk of data, and if it is necessary to close the back end connection immediately, the connection to the back end is closed before an attempt is made to write the data to a possibly slow front end HTTP client. This saves a significant amount of resources on back end servers, which no longer have to wait for a slow HTTP client to consume the data.

- Closing connections

If keepalives are enabled and not specifically disallowed by the request, the socket is left open in case a subsequent request needs it. This socket is registered with the connection pool cleanup, so that if no further requests to this back end are required, the socket will be closed cleanly.

The possibility still exists though that keepalive connections may be left open to back end servers when a slow client has requested other data. This will cause connections to remain open to back end servers, which can potentially be a performance problem. Thus it can sometimes be a performance gain to switch off keepalives to the back end servers.

- HTTP Pipelining

The HTTP protocol supports the concept of **pipelining**, which allows an HTTP client to send requests one after the other in one go, rather than for waiting for each response in turn. The protocol guarantees that responses are returned in the same order as the original requests.

Apache can support pipelining between the HTTP client and the apache server as the server only reads data in predetermined quantities, however pipelining cannot be supported across the proxy. This is due to a limitation in the filter stack code and the Apache architecture. It is currently not possible to read from more than one filter stack simultaneously, as would be required in the pipelining case to read requests from the HTTP client and the response from the back end server at the same time. In addition to this, changes would have to be made to the Apache architecture so that a second request could be read and passed to the required handler while the first request was still being handled.

- Filter Stacks

The introduction of the filter stack mechanism to Apache v2.0 has allowed content from handlers such as mod_proxy and mod_cgi to be modified on the fly before being sent to the HTTP client, which was not possible in Apache v1.3.

This is best demonstrated by content filters such as compression filters mod_gzip and mod_deflate, which now make it possible to compress data before being sent to the network, regardless of which handler within Apache was used to generate the data in the first place.

This allows Apache to perform tasks such as compression, content filtering with modules like mod_include, and even possibilities such as language translation.

How an FTP request is handled

Proxy can also handle ftp requests via the proxy_ftp module. This module would typically be used in forward proxy application, but could also be used in a reverse proxy to make an ftp server accessible via a web server.

- Connections

Operation of the ftp module handles back end connections in a similar fashion to the http module. The same DNS round robin behaviour applies. Once a connection has been established, either a file is downloaded, or a directory listing is obtained depending on the URL specified.

- Transfers

IPv6 compatible EPSV, passive PASV, and then standard PORT connections are attempted in turn to the remote server. This removes the need for a setting to specify which technique should be used.

- FTP directory listings

Pretty printing of FTP directory listings is achieved using a special filter that is inserted into the front filter stack when a directory listing is detected. The handler code renders a directory listing in a specific line-by-line format, which is parsed by the filter.

If the ftp server provided a message on the control channel (usually a README file), it is passed via an entry in the request notes table and incorporated into the filtered output.

- Reusing FTP connections

Currently the reusing of ftp connections is not currently implemented.

How a CONNECT request is handled

The proxy_connect module is significantly simpler than the other modules above.

After receiving a request line for a CONNECT, the module makes an attempt to connect to the remote server as done in proxy_http and proxy_ftp.

If this succeeds, the connect module listens simultaneously for traffic from either the back end server or the HTTP client, and passes any data it receives as is, behaving as a tunnel. The request is complete when either of the links is broken.

Future Enhancements to mod_proxy

There are a number of places within proxy where the behaviour can be improved or made more efficient.

- Better handling of back end connections

The current round robin behaviour is simplistic, although it does currently support the idea of trying other servers where it gets a connection refused.

If a server allows connections but is very slow in responding, there is no mechanism for detecting this condition, and then subsequently avoiding that server, or even flagging the status of the server for an administrator's attention.

A placeholder exists in the proxy_http code where such a hook could be inserted to determine the best server to connect to.

- Dynamically responding to server load

It would be useful for a mechanism to be created to calculate the load on back end servers (possibly through a measurement of server response times) and to shift load to underutilised servers.

- Ability to specify multiple servers

Presently multiple back end servers need to be registered under a single DNS name, either in the DNS file or in /etc/hosts. It would be useful to be able to specify multiple servers within the URL as is possible in LDAP URLs, such as this:

```
ProxyPass /foo/ http://server1 server2 server3/foo/
```

- Session Persistence

In some applications it is necessary for all hits from a specific HTTP client to be directed to the same back end server of many back end servers, in order for server side persistence to be possible without the need for back end servers to share their session data.

This can be achieved by using a client side cookie to distinguish between sessions, to use part of the URL prefix or query string, or to use a hash of the HTTP client's IP address.

This could be added as a possible hook to the connection code discussed above.

- Pipelining

It would be nice to support pipelining, however this is not likely to be practical under the current Apache architecture.

- Supporting more protocols (LDAP?)

Support for further protocols such as SMTP or LDAP could be made possible through the addition of new proxy protocol modules. It would have to determined first whether some of these protocols would be useful to implement.

- Cookie rewriting

In the present proxy, the Location and Content-Location headers are rewritten as they pass through the proxy to ensure the back end URL referenced in the response is mapped to the front end URL as seen by the HTTP client.

The scope of cookies, however, is currently not rewritten, as it is not always clear whether mapping the scope is the correct thing to do.

An example of where it might be a problem is where the back end specifies a scope of / meaning "be valid for the entire URL space", while the front end maps it to a sub-URL, like so:

```
ProxyPass /login/app/ http://server1/
```

The workaround to this problem is to ensure the back end URL space and the front end mapped URL space are the same, i.e.:

```
ProxyPass /login/app/ http://server1/login/app/
```

A better solution to this problem would be preferable.

- Configuring the Back end Filter Stack

Initially the back end filter stack was hard coded to include the basic network filters, and it was assumed that any filtering needed on a request would be done on the frontside filter stack.

There have been some applications, such as header rewriting, which have required filtering on the back end filter stack specifically. A proper framework for adding arbitrary filters to the back end filter stack is needed, preferably in as similar a way to adding to the front end filter stack as possible.

Conclusion

Mod_proxy as implemented in httpd 2.0 provides not only a reference implementation of an RFC 2616 proxy, but also a useful set of functions for webmasters in real world situations.

The most recent version of this paper is available at:

http://www.topsail.org/goodies/