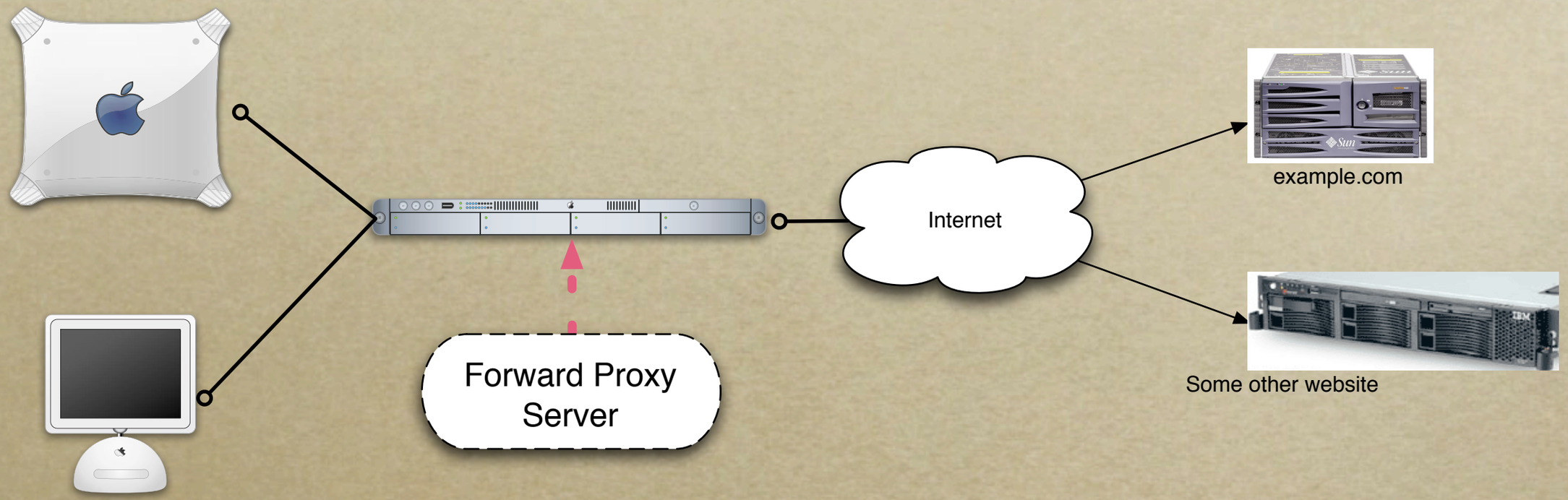
Painless Web Proxying with Apache mod_proxy

Justin R. Erenkrantz
University of California, Irvine and Google, Inc.
http://www.erenkrantz.com/oscon/
justin@erenkrantz.com

Why should I pay attention?

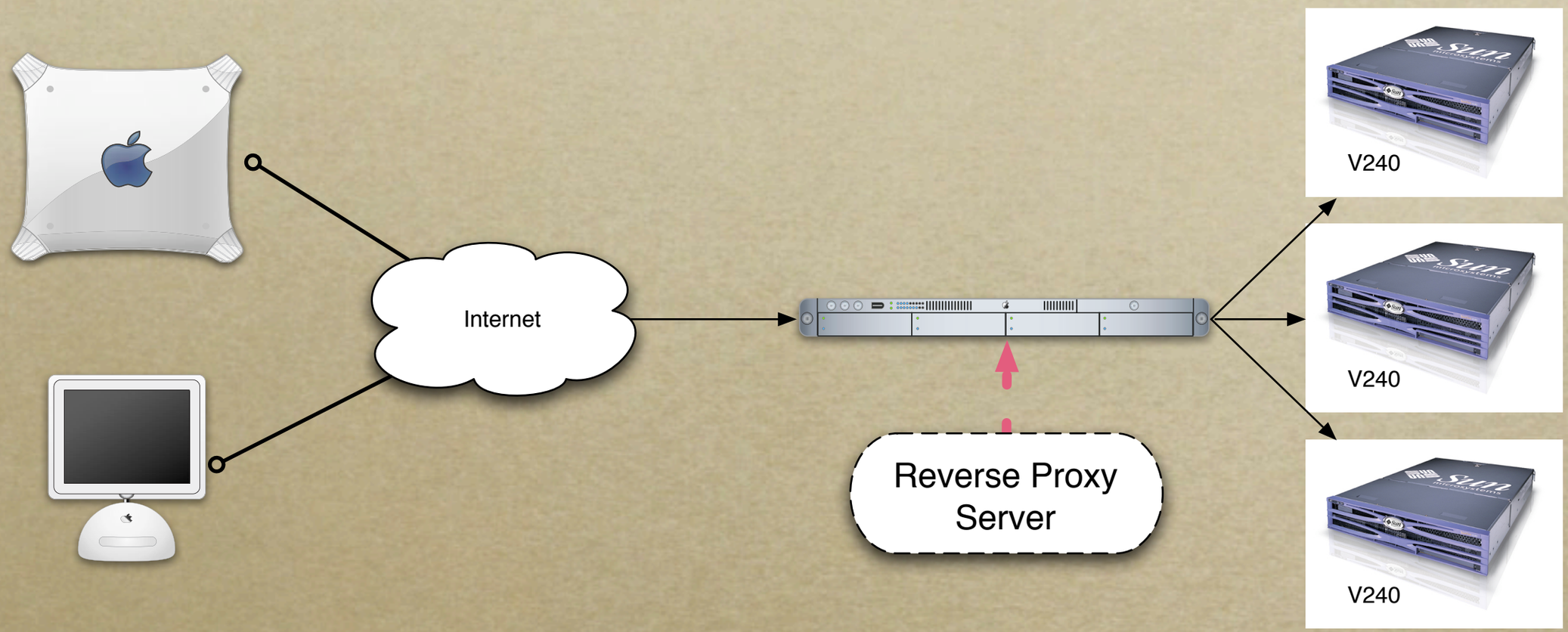
- *Apache HTTP Server committer since 2001*
- *Also involved with APR and Subversion*
- *Director, Apache Software Foundation*
- *Engineering Intern, Google, Inc.*
- *Ph.D. student at UC Irvine*

Forward Proxy



- *Multiple clients route all HTTP traffic through the same outgoing server*

Reverse Proxy / Gateway



- *Distributes incoming requests to multiple "identical" backends*

mod_proxy Goals

- *Squid is a fully featured HTTP forward proxy cache - suitable for a workgroup, etc.*
- *mod_proxy + mod_cache can do a passable job as a forward caching proxy, but that hasn't been a main focus of recent work*
- *Recent work on mod_proxy is aimed at serving the needs of reverse proxies*

What is Apache HTTP Server?

- *Derived from NCSA httpd*
- *"Best web server money can't buy"*
- *Over 60% market share - Netcraft*
- *Extensible, modular architecture*
- *Has built-in support for forward and reverse proxies*

Apache HTTP Server History

- *1.0 released in December, 1995*
- *1.3.0 released in June, 1998*
- *2.0.35 (first 2.0 GA) released April, 2002*
- *2.2.0 released December, 2005*
- *2.2.2 released May, 2006*

Overview of 2.x Features

- *2.0 was a major architectural rewrite*
- *Provides a solid platform for further work*
- *Introduction of APR abstracts OS details*
- *Threadable MPMs = lower memory footprint*
- *Filters bridge a long-time gap present in 1.x*
- *IPv6 support, mod_ssl & mod_dav bundled*

mod_proxy's History

- *mod_proxy has an unusual history*
  - *First included in 1.1 (circa 1996)*
  - *Punted when 2.0 started as it was broken*
  - *Came back just in time for 2.0 GA*
  - *Rewritten for 2.2 with load balancing*
- *Is the third time the charm? We hope.*

mod_proxy Supported Protocols

- *HTTP/0.9, HTTP/1.0, HTTP/1.1*
- *SSL traffic via mod_ssl*
- *AJP13 - Tomcat's mod_jk protocol*
- *FTP (only supports GET)*
- *CONNECT (SSL Proxying)*
- *FastCGI support currently in trunk (2.3+)*

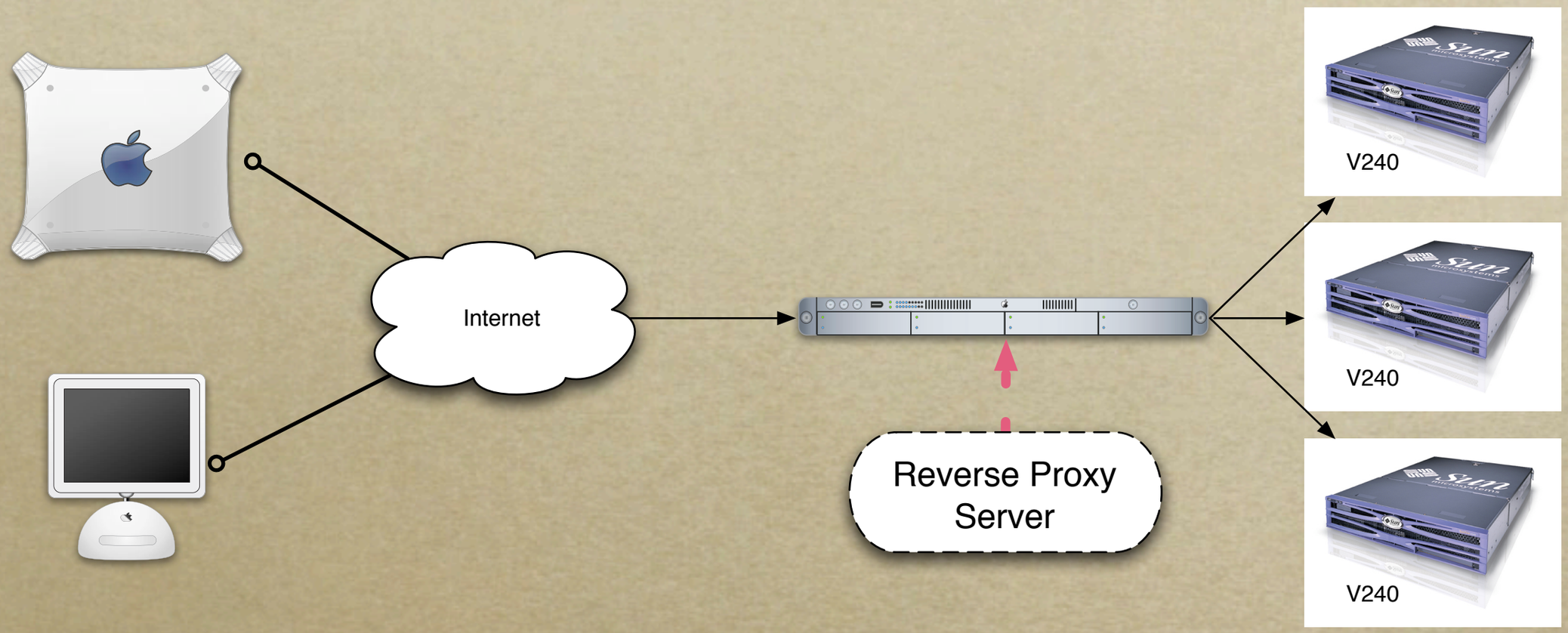
Configuring a Forward Proxy

Listen 3128
ProxyRequests on
*<Proxy *>*
Order Deny,Allow
Deny from all
Allow from 192.168.0
</Proxy>

Then, configure your browser to use http://proxy.example.com:3128/

All requests made by your local browser will be relayed to the forward proxy. No direct connections to the outside world will be made.

What is load balancing?



- *Distribute load to several ‘identical’ servers*
- *Gateways are transparent to external users*

Load balancing in mod_proxy

- *Pluggable module to balance load across all available backends: mod_proxy_balancer*
  - *Request counting (round-robin)*
  - *Weighted traffic average (based on bytes)*
- *A user can be "sticky" to a backend based on cookies (JSESSIONID, PHPSESSIONID)*

Backend optimizations

- *mod_proxy supports connection pooling*
  - *Connections are shared within a process*
    - *Useful if using worker or event MPM*
  - *Backends can be added or removed while the system is online through the balancer-manager interface*

Balancer Manager

http://localhost:8080/balancer-manager/?b=example&s=http&w=server1

Load Balancer Manager for localhost

Server Version: Apache/2.1.7-dev (Unix) mod_ssl/2.1.7-dev OpenSSL/0.9.7b DAV/2
Server Built: Jul 30 2005 13:39:56

LoadBalancer Status for balancer://example

StickySession	Timeout	FailoverAttempts	Method
	0	2	Requests

Scheme	Host	Route	RouteRedir	Factor	Status
http	server1			1	Ok
http	server2			1	Ok
http	server3			1	Ok

Edit balancer settings for balancer://example

StickySession Identifier:

Timeout: 0

Failover Attempts: 2

LB Method: Requests

Submit

Done

Reverse Proxy Example

Connection reuse:

ProxyPass /example http://backend.example.com min=0 max=20 smax=5
ttl=120 retry=300
ProxyPassReverse /example http://backend.example.com/

Option	Description	Default
min	*Minimum number of connections to keep open*	0
max	*Maximum connections to keep open to server*	*1 or n**
smax	*(Soft maximum) Try to keep this many connections open*	***max***
ttl	*Time to live for each connection above smax*	*none*
retry	*If conn. fails, wait this long before reopening conn.*	*60 sec*

** = If threaded MPM, use ThreadsPerChild; otherwise 1*

Serve from front-end directly

ProxyPass /images !
ProxyPass /css !
ProxyPass / http://backend.example.com

- *ProxyPass with ! is useful when you have static content (images, CSS, etc.)*
- *Avoids the overhead of going to the backend*

AJP / mod_proxy_ajp

- *httpd 2.2+ can talk to Tomcat natively!*
  - *Built-in bundled module to replace mod_jk*
    - *No external modules needed with 2.2*
  - *Configure Tomcat to listen on the AJP port*
  - *Set up Tomcat like you would with mod_jk*

mod_proxy AJP Example

ProxyPass / balancer://example/
<Proxy balancer://example/>
*BalancerMember **ajp**://server1/*
*BalancerMember **ajp**://server2/*
*BalancerMember **ajp**://server3/*
</Proxy>

The only difference is we replace http with ajp.
mod_proxy and mod_proxy_ajp does the rest.

FastCGI

- *Usually recommended with Ruby on Rails*
- *Avoids the overhead of spawning new CGI processes on each request*
- *FastCGI daemon with a custom binary protocol listener on port 8000*
- *Only in httpd's trunk (2.3+) - may be backported to 2.2.x, but not sure yet.*

mod_proxy FastCGI Example

ProxyPass / balancer://example/
<Proxy balancer://example/>
*BalancerMember **fcgi**://server1/*
*BalancerMember **fcgi**://server2/*
*BalancerMember **fcgi**://server3/*
</Proxy>

Again, the only difference is we replace http with fcgi.

Your own protocol handler...

- *FTP and other protocols also supported by just replacing the URI scheme*
- *What if you want to create your own protocol handler for mod_proxy?*
- *It's not that bad...almost.*
- *Let's use mod_proxy_fcgi as our example...*

Walking tour of mod_proxy_fcgi

mod_proxy_fcgi is a good example to learn from because it has been written relatively recently and can take clear advantage of the new features of mod_proxy.

It will just send the requests to the FastCGI daemon and receive a response.

Source:
http://svn.apache.org/repos/asf/httpd/httpd/trunk/modules/proxy/mod_proxy_fcgi.c

% wc -l mod_proxy_fcgi.c
998 mod_proxy_fcgi.c

Exists only in trunk. mod_proxy_fcgi be backported to 2.2.x in the future...

All of the logic responsible for talking to FastCGI is self-contained to this one module and one file.

Apache module terminology

- *Directive: Configuration syntax (httpd.conf)*
- *Hooks: Code run at a certain point during request lifecycle*
- *Filters: Transformation of data: in and out*
- *Bucket brigades: Streams of "Bits"*
- *Handlers: Generation of data*

mod_proxy_fcgi is a handler, but it interacts with all of the above

Four main steps for a reverse proxy

1. *Determine which backend to direct request to*
2. *Make/reuse connection to the backend*
3. *Process the request and deliver the backend response - fcgi_do_request()*
4. *Release the connection*

Steps 1, 2, and 4 use common mod_proxy code;
Only step 3 is customized for FastCGI...

fcgi_do_request() phases

- *Tell FastCGI we're starting a request...*
- *Send the CGI environment*
- *Calls dispatch() to handle request/response*
  - *Pass along the request headers and body*
  - *...wait...*
  - *Read the response headers and body*

Dealing with Brigades

- *Apache 2.x deals with "bucket brigades"*
- *Brigades are collections of buckets*
- *Buckets are a "chunk" of data*
- *Handler - more specifically, dispatch() - creates bucket brigades and passes them down the filter chain (and also reads buckets for input too)*

Passing along the request body

On our connection to the backend, we'll do a loop around apr_poll() to wait until it's safe to write without blocking...

rv = apr_poll(&pfd, 1, &n, timeout);

...

if (pfd.rtnevents & APR_POLLOUT) {

Now, we'll read data from the request body via input filters:

rv = ap_get_brigade(r->input_filters, ib, AP_MODE_READBYTES,
APR_BLOCK_READ, sizeof(writebuf));

...we'll format the ib brigade's buckets into a flat iovec structure...

Pass the data to the FastCGI daemon using send_data():

rv = send_data(conn, vec, 2, &len, 0);

Processing the response

Inside the same apr_poll() loop, we'll wait until we should read:

```
if (pfd.rtnevents & APR_POLLIN) {
```

Read the data from FastCGI using get_data helper:

```
rv = get_data(conn, (char *) farray, &readbuflen);
```

Translate the data into buckets:

```
b = apr_bucket_transient_create(readbuf, readbuflen, c->bucket_alloc);
APR_BRIGADE_INSERT_TAIL(ob, b);
```

Pass the ob brigade onto the output filters so it can be sent to client:

```
rv = ap_pass_brigade(r->output_filters, ob);
```

Note how HTTP headers are handled!
They must be set before first body byte is sent down the output filter chain or they will not be sent to the client.

Caching with mod_proxy

Transparently cache from backend and store it on the local disk:
CacheRoot /var/cache/apache/
CacheEnable disk /

If the cache can not satisfy the request, it'll process the request normally - i.e. contact the reverse proxy.

Use htcacheclean to control the size of the on-disk cache:
htcacheclean -d15 -p/var/cache/apache -l250M
(Every fifteen minutes, ensure the cache is no bigger than 250MB.)

For more information about caching:
http://httpd.apache.org/docs/2.2/caching.html

Recap

- *mod_proxy supports a variety of protocols*
  - *HTTP, HTTPS, AJP, FastCGI, FTP...*
- *Can act as a forward or reverse proxy*
- *2.2+ features built-in load balancing*
- *Examples of how a backend provider is written using the mod_proxy framework*

Painless Web Proxying with Apache mod_proxy

Justin R. Erenkrantz
University of California, Irvine and Google, Inc.
http://www.erenkrantz.com/oscon/
justin@erenkrantz.com