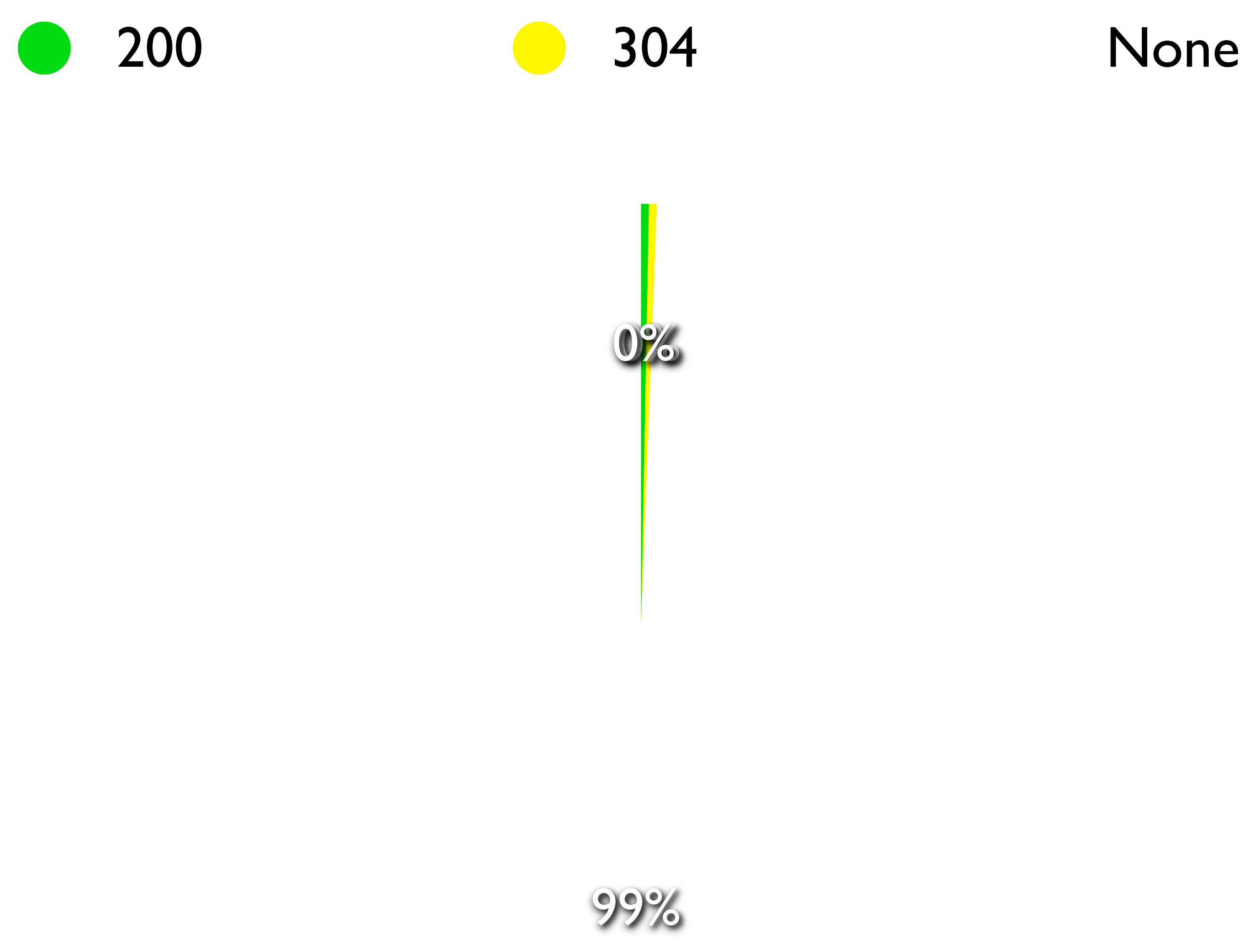
Policing the RFC:

How to not kill your website at scale

Graham Leggett

Apachecon EU 2012



Caching is Good

Tuesday 06 November 12

Start with an aside, caching is good.

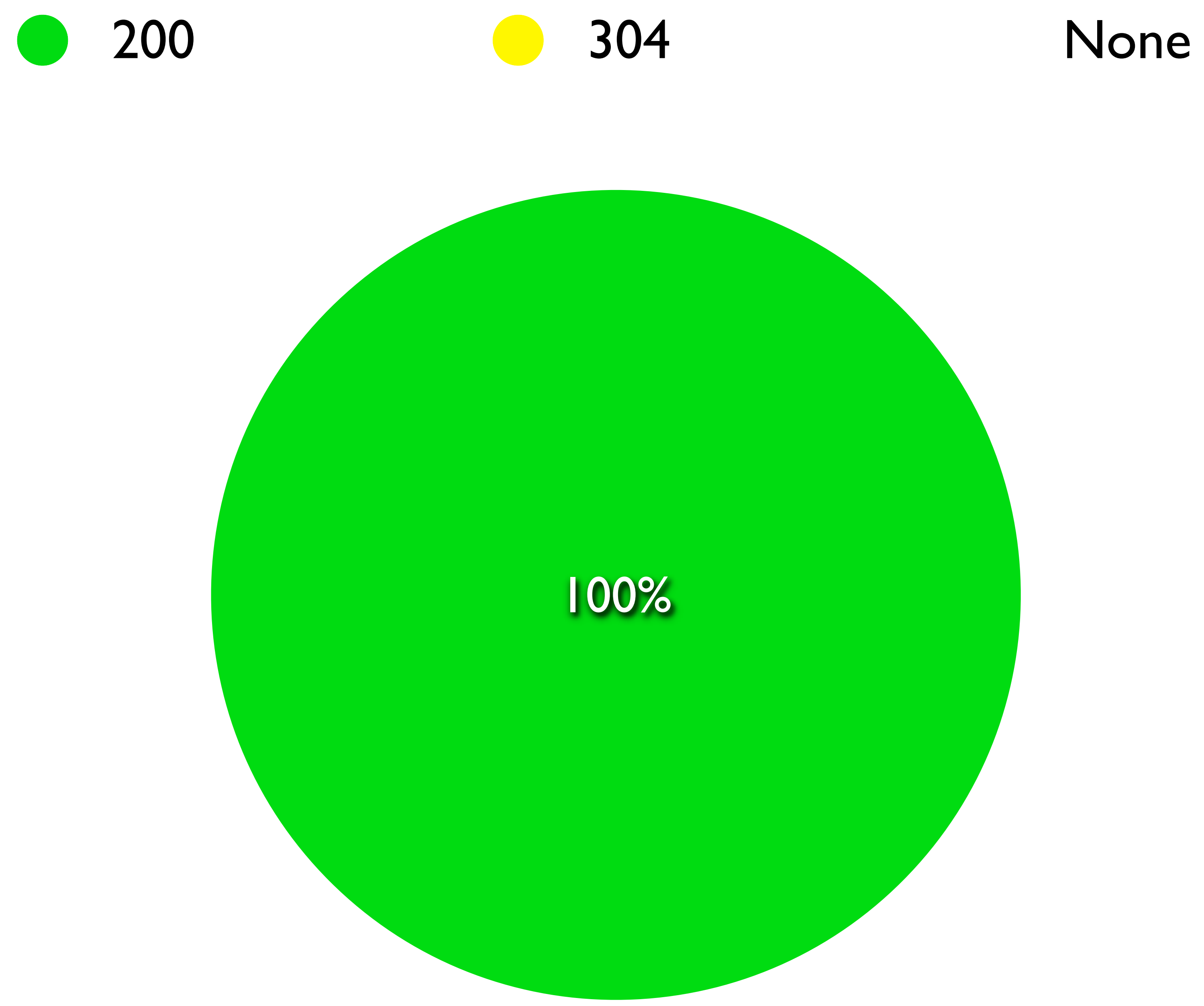
In a rough non-scientific test, we compared highly cacheable elements to their non cacheable parent page, and found that in this particular instance 99.4% of the traffic was never seen by the servers at all.

0.3% of the traffic consisted of cheap 304 Not Modified responses.

0.3% of the traffic consisted of 200 OK.

You want to cache your traffic as much as you can.

You want to support conditional requests - in this example, traffic was half.



No Caching is Bad

Tuesday 06 November 12

Risks of a sudden loss of caching:

- Flatten your infrastructure
- Get a massive bill

http://tools.ietf.org/html/rfc1122#section-1.2.2

1.2.2 Robustness Principle

At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability [IP:1]:

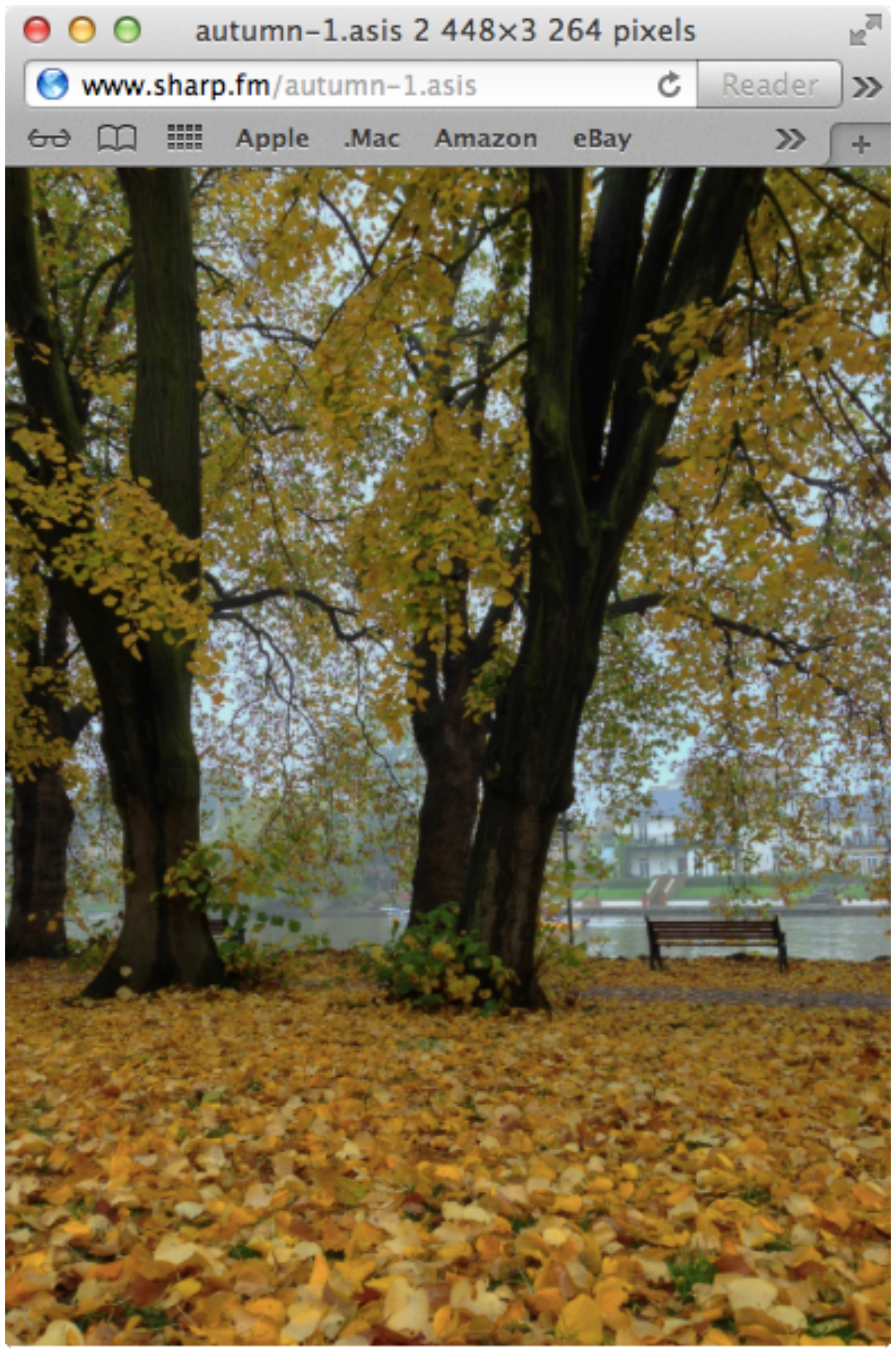
"Be liberal in what you accept, and conservative in what you send"

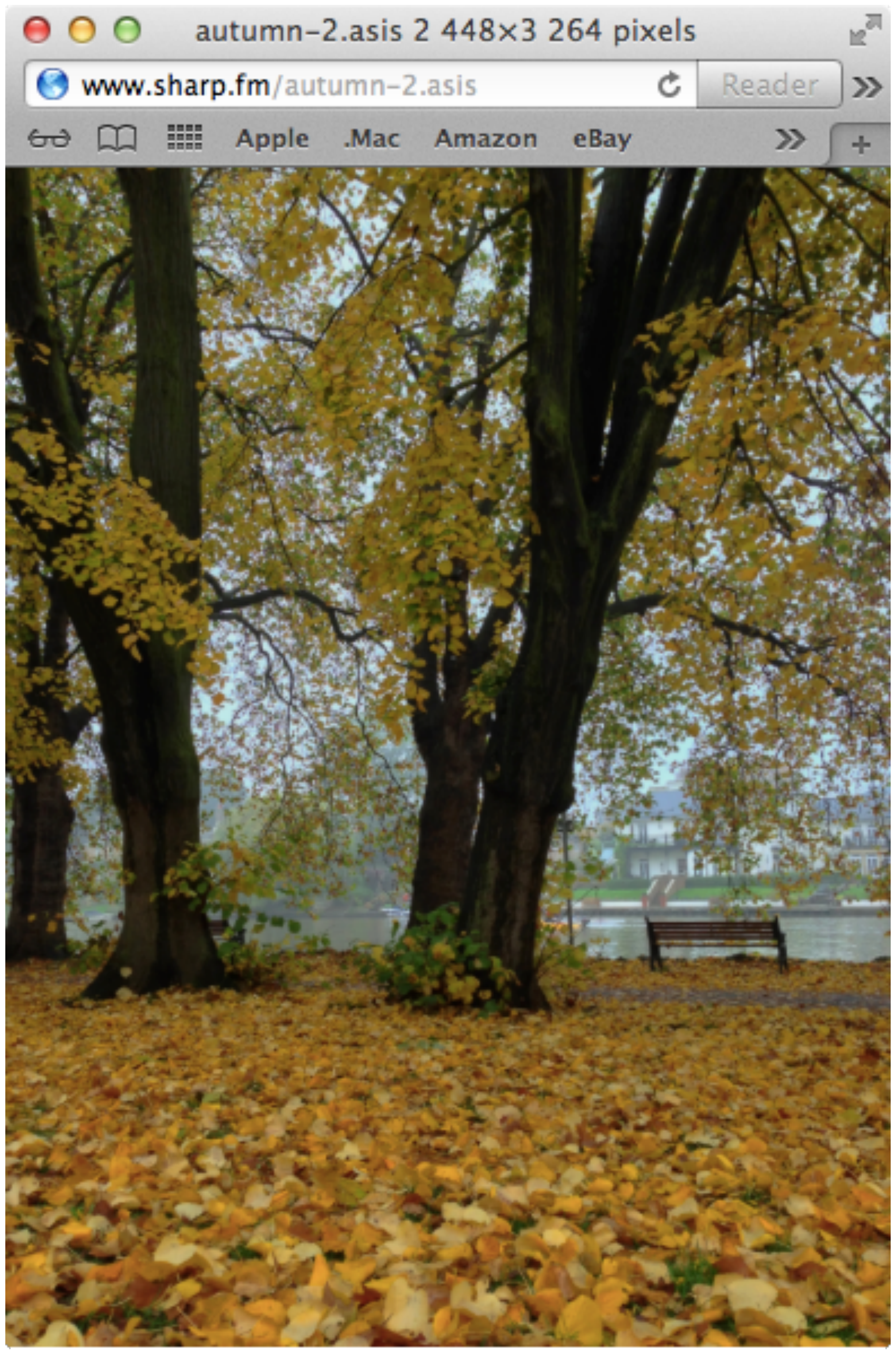
RFC1122 says “Be liberal in what you accept, and conservative in what you send”.

It means that servers are strict, and clients are forgiving.

We do our testing with clients.

Problem.





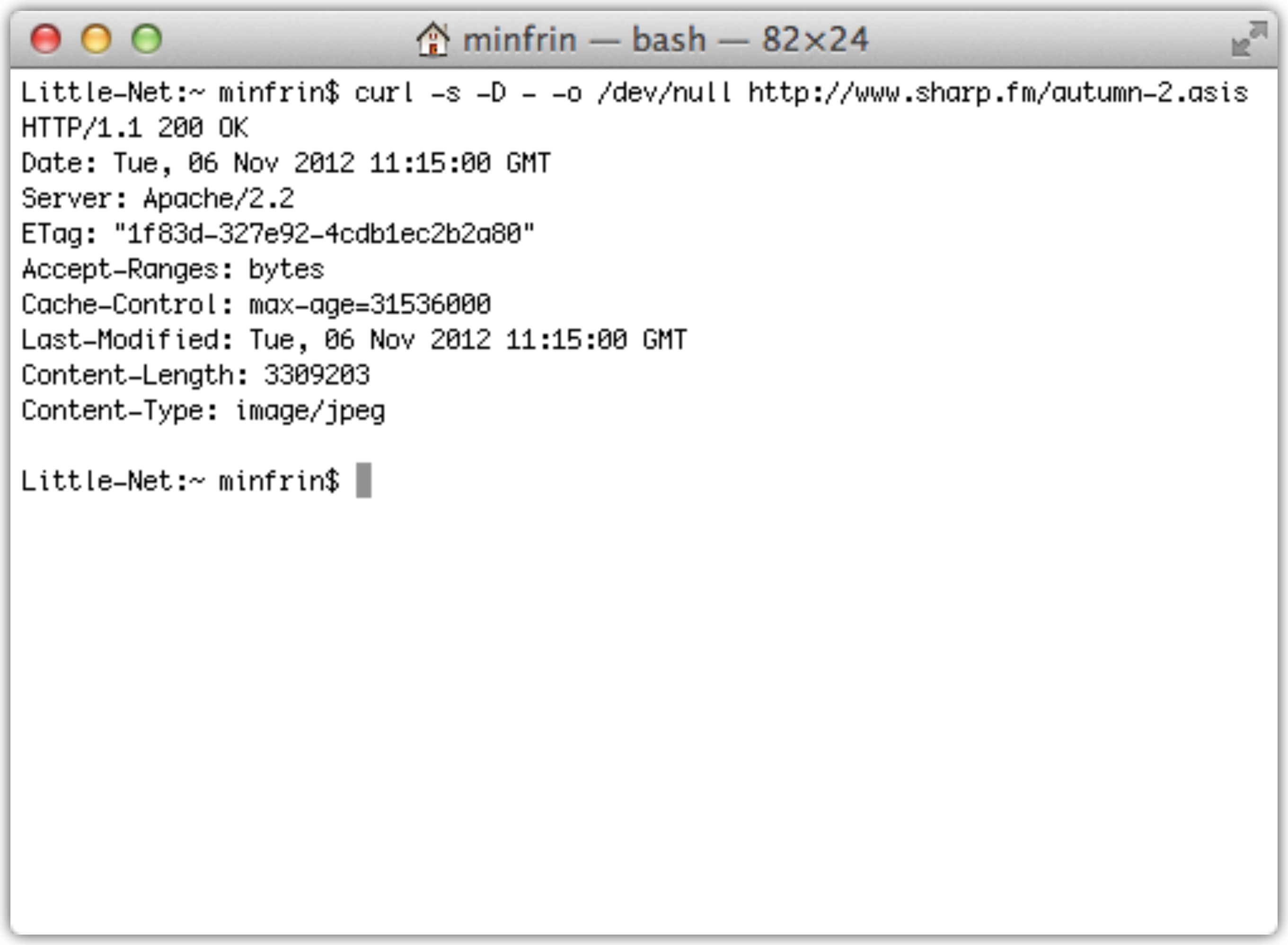
Tuesday 06 November 12

Concrete example.

One of these pictures is RFC compliant.

One of these pictures will kill your site.

Autumn 2



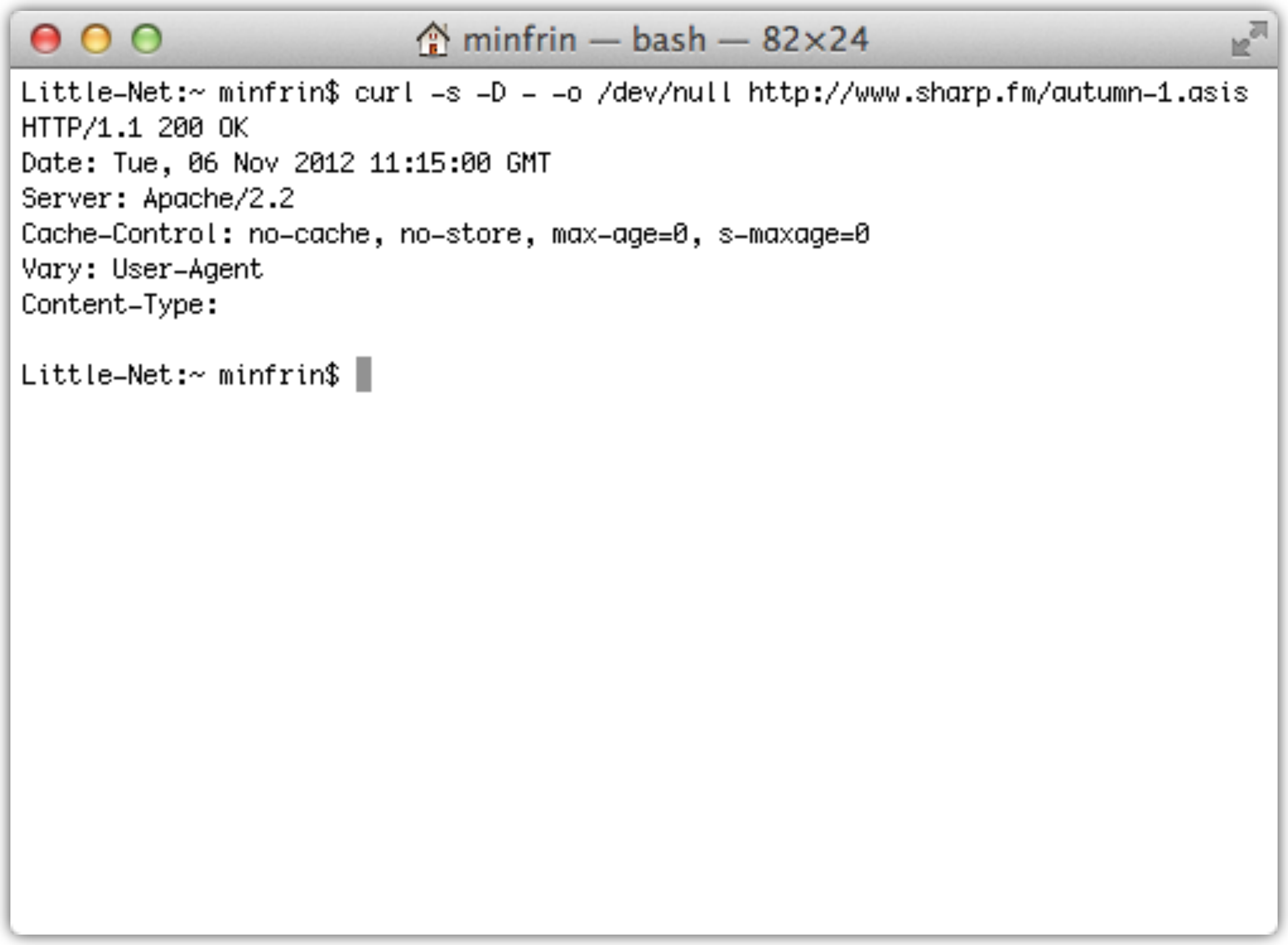
```
Little-Net:~ minfrin$ curl -s -D - -o /dev/null http://www.sharp.fm/autumn-2.asis
HTTP/1.1 200 OK
Date: Tue, 06 Nov 2012 11:15:00 GMT
Server: Apache/2.2
ETag: "1f83d-327e92-4cdb1ec2b2a80"
Accept-Ranges: bytes
Cache-Control: max-age=31536000
Last-Modified: Tue, 06 Nov 2012 11:15:00 GMT
Content-Length: 3309203
Content-Type: image/jpeg

Little-Net:~ minfrin$
```

Tuesday 06 November 12

Autumn 2 has:

- Proper Etags, revalidation results in cheap 304 Not Modified
- A well defined long expiry
- Has content length, a cache can decide up front whether to cache
- Has a valid content type, filters will be correctly applied

Autumn 1



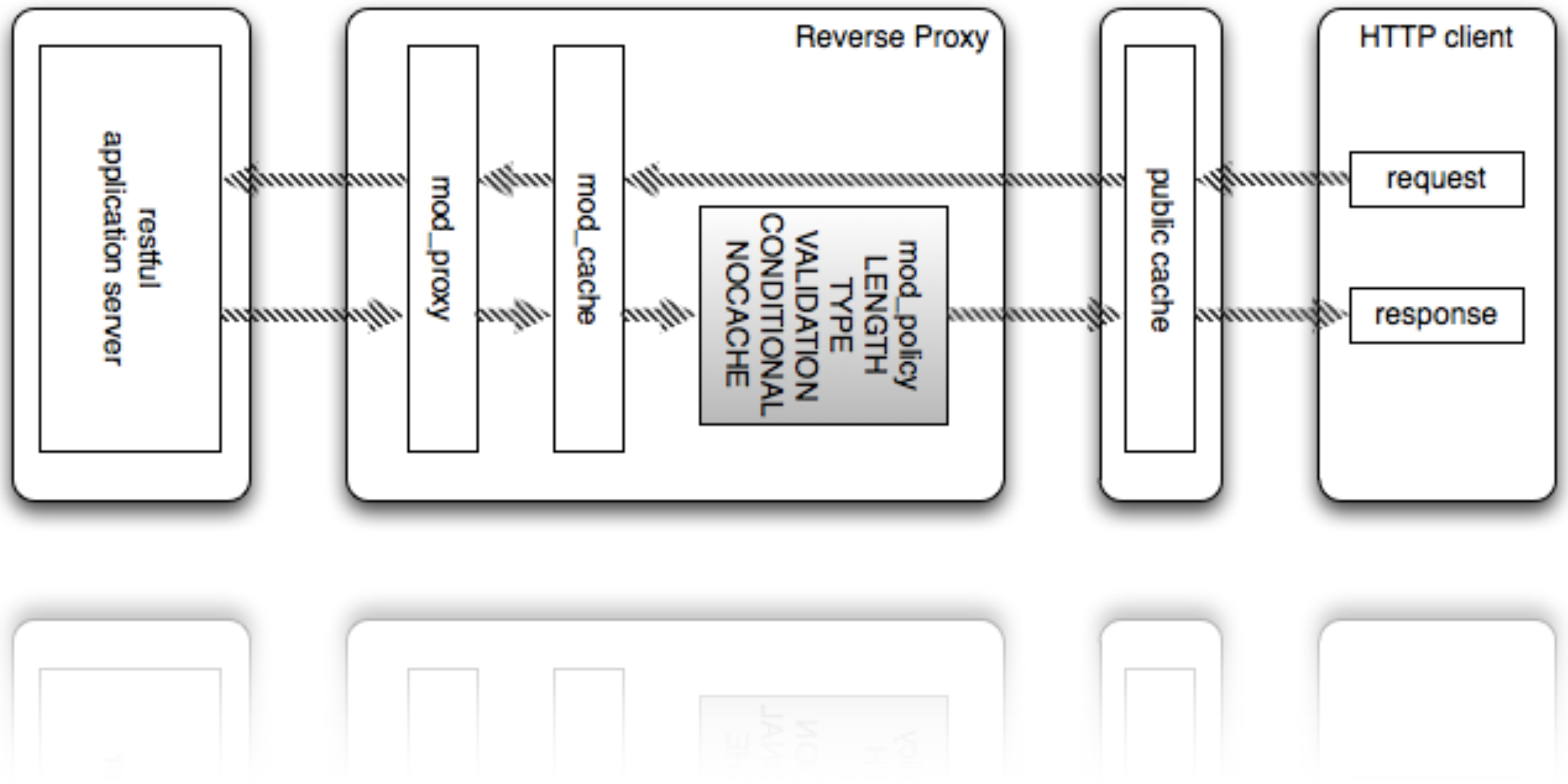
```
Little-Net:~ minfrin$ curl -s -D - -o /dev/null http://www.sharp.fm/autumn-1.asis
HTTP/1.1 200 OK
Date: Tue, 06 Nov 2012 11:15:00 GMT
Server: Apache/2.2
Cache-Control: no-cache, no-store, max-age=0, s-maxage=0
Vary: User-Agent
Content-Type:

Little-Net:~ minfrin$
```

Tuesday 06 November 12

Autumn 1 has:

- No ETag or Last-Modified, revalidation impossible
- No cache / cache bypass
- Vary on the User-Agent, a cache DoS
- Content-Length is missing, some caches won’t cache this
- Content-Type is blank, filters are bypassed

What can we do?



mod_policy

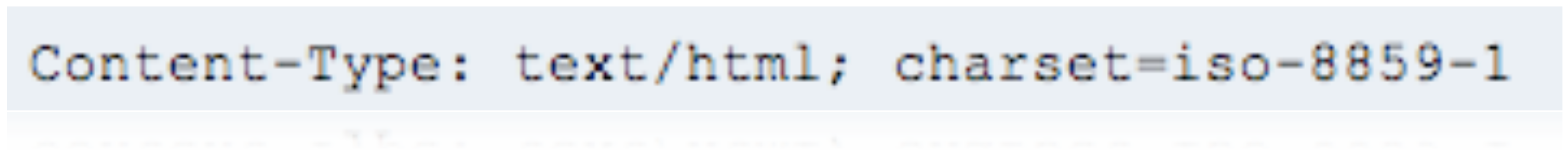
How do we enforce protocol compliance?

mod_policy has been designed to enforce a specified protocol policy.

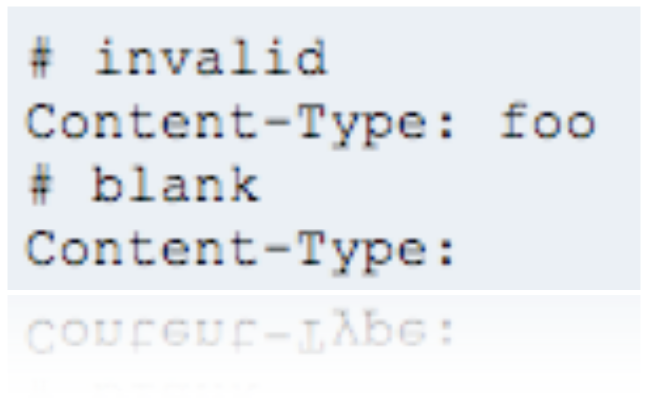
It provides a set of output filters, which are slotted into place as needed.

We can log the violation, or we can reject the request outright with a 500 error.

While mod_security protects us from clients, mod_policy protects us from servers.



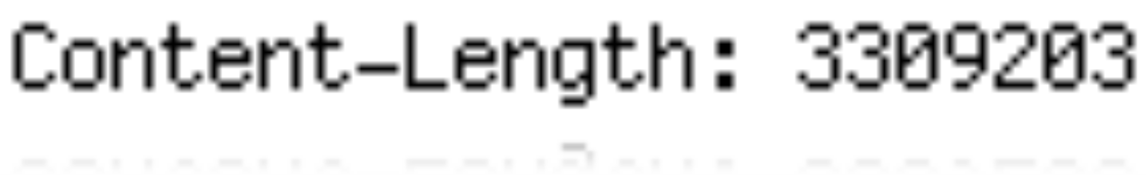
```
Content-Type: text/html; charset=iso-8859-1
```



```
# invalid
Content-Type: foo
# blank
Content-Type:
```

Content Type Policy

The most basic check, do we have a content type at all, is the content type the content type we were expecting?

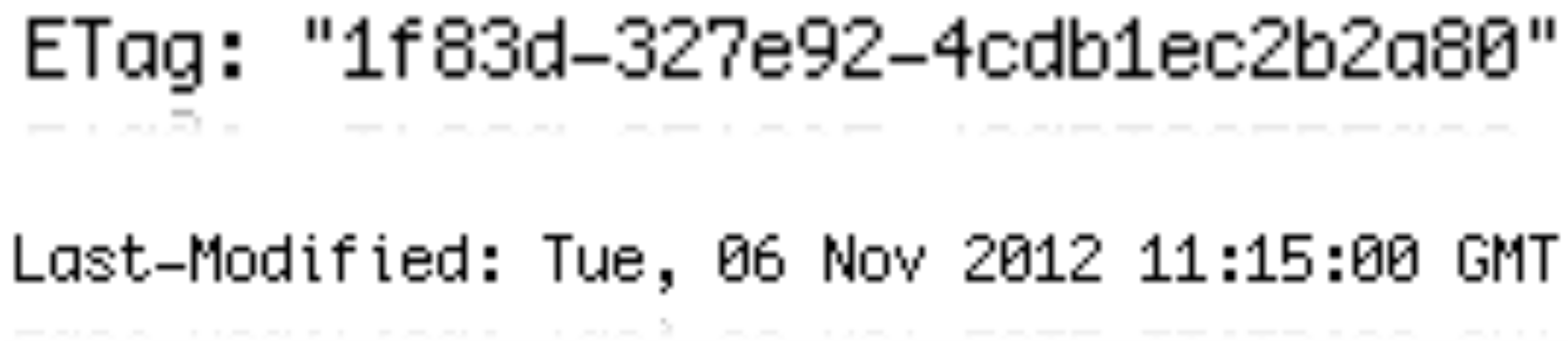


```
Content-Length: 3309203
```

Content Length Policy

Some caches want advance notice of the object size before the cache will attempt caching.

We can enforce the presence of a content length.

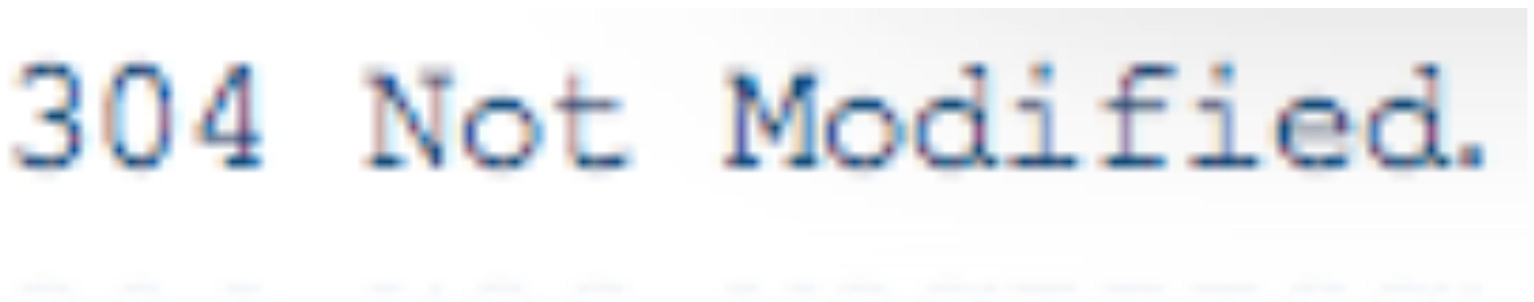


```
ETag: "1f83d-327e92-4cdb1ec2b2a80"

Last-Modified: Tue, 06 Nov 2012 11:15:00 GMT
```

Validation Policy

Is an ETag or Last-Modified present?

Is the ETag well formed? Does the ETag have quotes around it? Is a weak ETag correctly specified? (W/””)



```
304 Not Modified.
```

Conditional Request Policy

Tuesday 06 November 12

Does the server honor conditional requests like it should?

If not, we can reject the request.

```
Cache-Control: max-age=31536000
```

```
Cache-Control: no-cache, no-store, max-age=0, s-maxage=0
```

Cache Maxage Policy

Tuesday 06 November 12

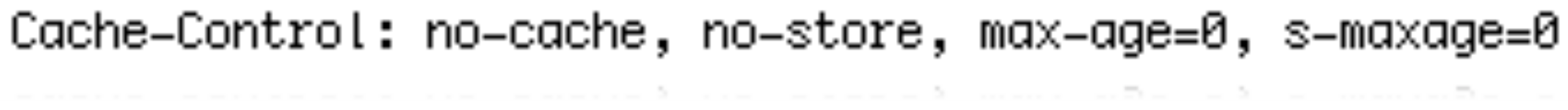
Here we zoom into the Freshness Lifetime of the response. Does the freshness lifetime fall below the acceptable minimum length of time?

A short freshness lifetime can amplify an outage.

For example, a freshness lifetime of 1 hour will cause all cached data to expire within an hour, and all caching will be gone after that point during a failure.

People may be tempted to place short freshness lifetimes on URLs in an effort to allow “take down” of content. In this case, simply abandon the URL.

First prize: cache forever.



```
Cache-Control: no-cache, no-store, max-age=0, s-maxage=0
```

No Cache Policy

Here we detect whether the server has completely banned caching altogether.

We accept the client sending no-cache, but we ban the server from responding no-cache.

This protects against thundering herds and bill shock.

```
Vary: User-Agent
```

Vary Policy

Tuesday 06 November 12

The Vary header tells proxies along the way which headers have been used to decide on the variant of the response being returned.

Vary on a header that has millions of possible values, and you could fill your cache with the same page, cached millions of times.

Example: User-Agent. In a rough test, after recording User-Agent strings for about 5 days, approximately 1 million unique User-Agent string combinations were recorded.

If an URL varies on User-Agent, it cause cause the cache to clog up with copies of the page, and become ineffective.

HTTP/1.1 200 OK

Keepalive / Version Policy

Tuesday 06 November 12

HTTP/1.0 requests that arrive can cause havoc with keepalive, potentially disabling keepalive when this is not intended.

This can cause problems with sockets in the CLOSE_WAIT and TIME_WAIT states.

As the exception to the rule, this filter can be used to ban HTTP/1.0 requests, insisting that clients use HTTP/1.1 as a minimum, where keepalive defaults to enabled.

With the addition of the Keepalive Policy, if keepalive is not present at all, the request can be rejected.

mod_cache

```
CacheHeader on
CacheDetailHeader on
```

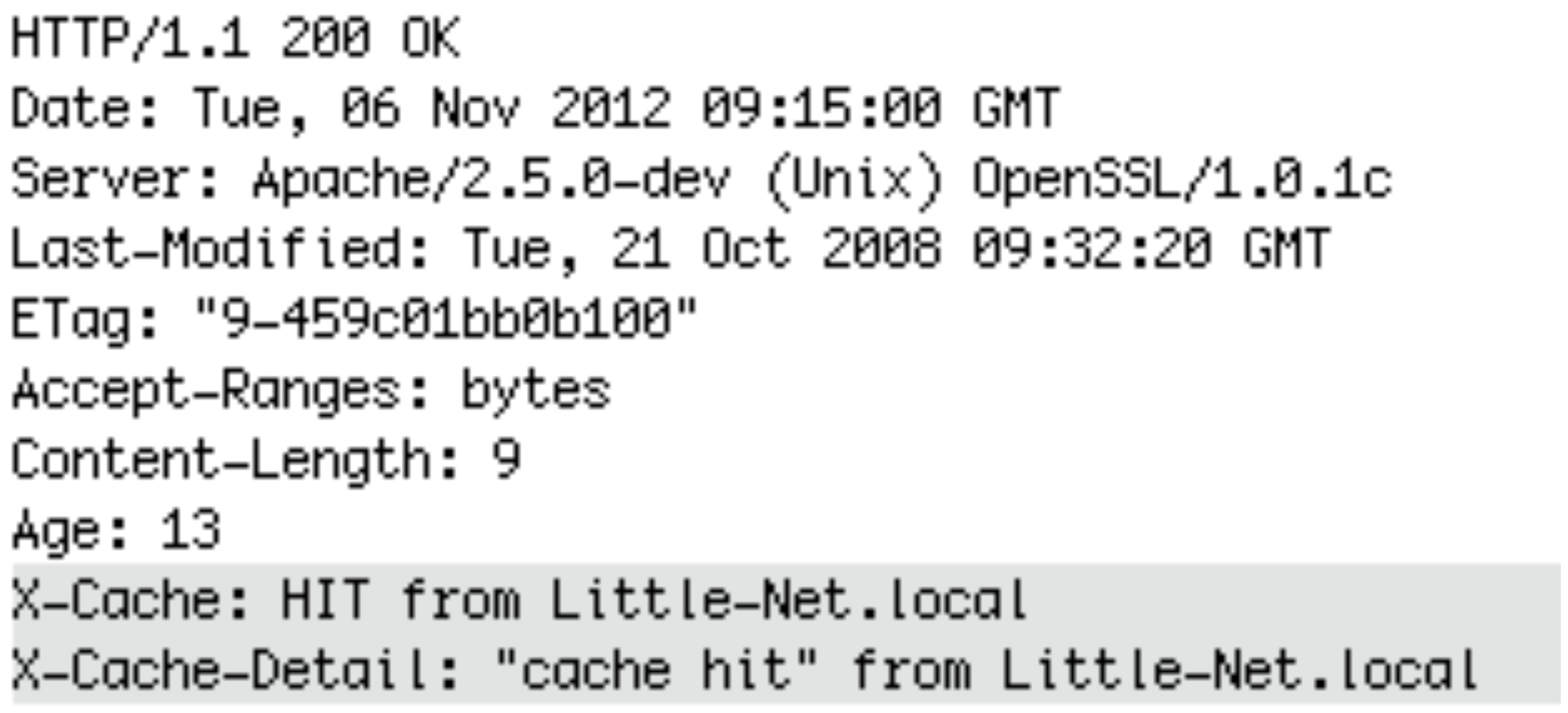
Tuesday 06 November 12

What caching edge cases might we find?

How can a developer dig deeper to determine what problems exist?

The CacheHeader and CacheDetailHeader directives give precise reasons for a caching decision.

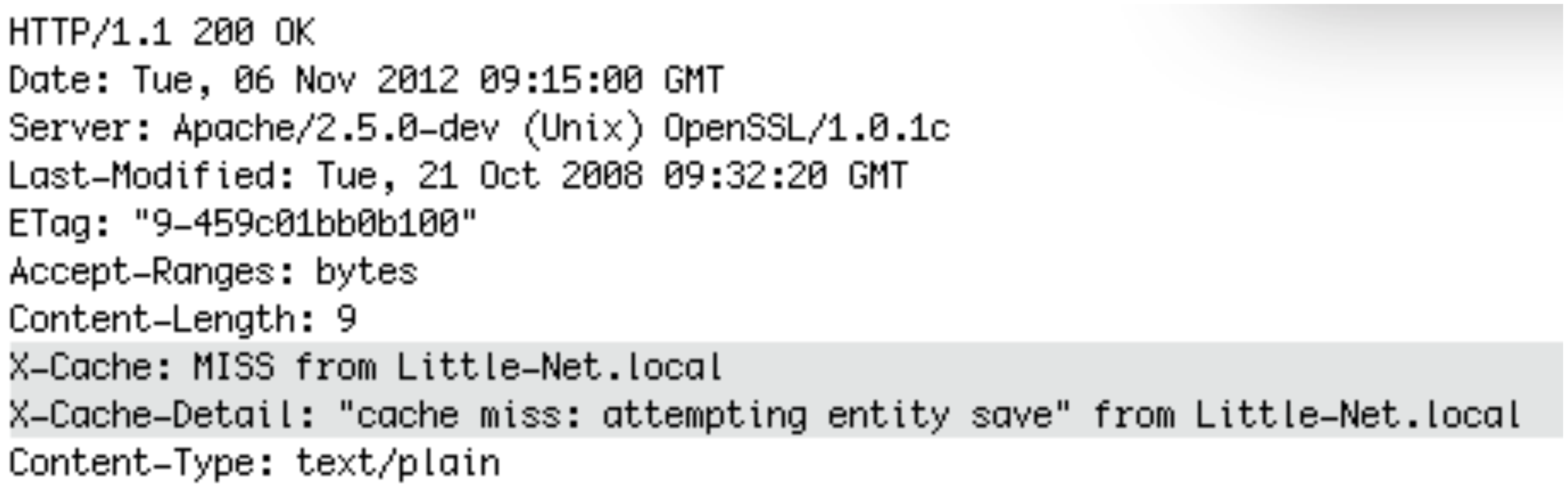
Cache Hit



```
HTTP/1.1 200 OK
Date: Tue, 06 Nov 2012 09:15:00 GMT
Server: Apache/2.5.0-dev (Unix) OpenSSL/1.0.1c
Last-Modified: Tue, 21 Oct 2008 09:32:20 GMT
ETag: "9-459c01bb0b100"
Accept-Ranges: bytes
Content-Length: 9
Age: 13
X-Cache: HIT from Little-Net.local
X-Cache-Detail: "cache hit" from Little-Net.local
```

In the standard cache case, we have a cache HIT reported.

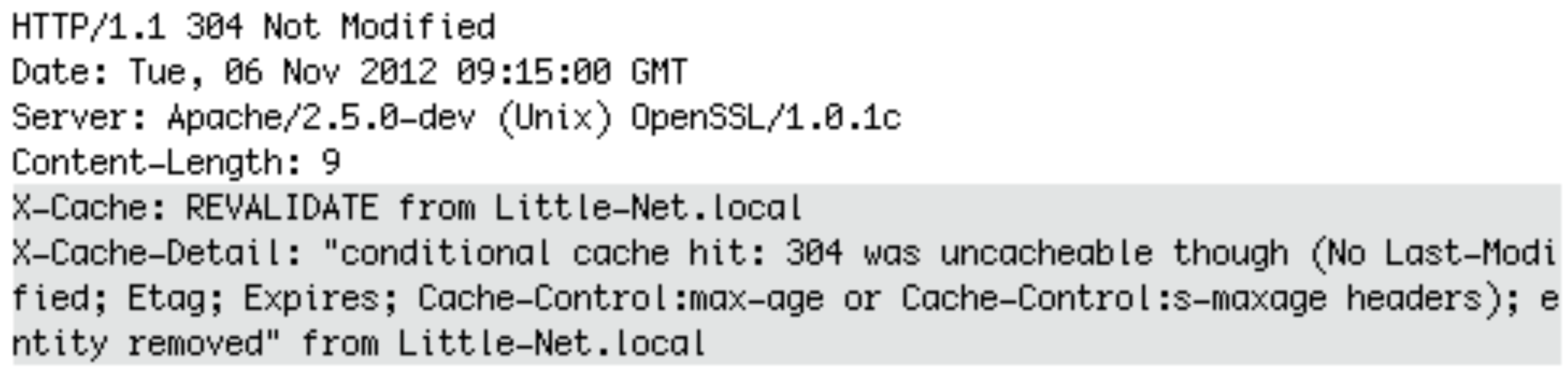
Cache Miss



```
HTTP/1.1 200 OK
Date: Tue, 06 Nov 2012 09:15:00 GMT
Server: Apache/2.5.0-dev (Unix) OpenSSL/1.0.1c
Last-Modified: Tue, 21 Oct 2008 09:32:20 GMT
ETag: "9-459c01bb0b100"
Accept-Ranges: bytes
Content-Length: 9
X-Cache: MISS from Little-Net.local
X-Cache-Detail: "cache miss: attempting entity save" from Little-Net.local
Content-Type: text/plain
```

In a further standard cache case, we have a typical cache MISS.

Cache Edge Case



```
HTTP/1.1 304 Not Modified
Date: Tue, 06 Nov 2012 09:15:00 GMT
Server: Apache/2.5.0-dev (Unix) OpenSSL/1.0.1c
Content-Length: 9
X-Cache: REVALIDATE from Little-Net.local
X-Cache-Detail: "conditional cache hit: 304 was uncacheable though (No Last-Modi
fied; Etag; Expires; Cache-Control:max-age or Cache-Control:s-maxage headers); e
ntity removed" from Little-Net.local
```

Here we have an edge case.

A previously cached entity is being revalidated, but the 304 Not Modified response is itself uncacheable, telling us the entity is no longer cacheable. The cache responds by honouring the response and removing the cached entry.

The next hit might cause the entity to be cached again, and the next attempt at revalidation will cause the entity to be removed again, and so on.

The symptom: the site runs slower during caching. The cache is blamed, but in reality the service behind it is not 100% compliant.

- Caching is good
- Sudden denial-of-caching is expensive
- You need to enforce RFC compliance
- mod_cache + mod_policy can help

- http://httpd.apache.org/docs/trunk/compliance.html
- http://people.apache.org/~minfrin/bbc-donated/mod_policy/
- http://httpd.apache.org/docs/2.4/mod/mod_cache.html#cachedetailheader