CacheMeter

by

Sagar Shailesh Sugandhi

(Under the direction of Dr. Kang Li)

Abstract

HTTP Response caching significantly reduces user perceived latency. This thesis describes CacheMeter, our tool which calculates caching percentage of webpage(s) with respect to browser caching. It calculates cacheability statistics for various content types and for HTTPS contents. We ran CacheMeter on multiple webpages for Alexa top 1000 websites. Results show that the average cacheability percentage per website received through HTTP(s) is 77%. These resources are potentially available for reuse from cache on subsequent requests. Results also show that larger objects have higher browser cacheability percentage, which meets expectation.

We later describe analysis of HTTP caching implementation in Firefox and its evolution through various releases. Firefox employs an aggressive approach by caching all objects, even when response has no caching headers. Firefox makes reuse decision on subsequent request. When a broken HTTP response is received, Firefox eliminates it from the cache.

Index words: HTTP caching, Web browser caching statistics, Firefox caching

CacheMeter

by

Sagar Shailesh Sugandhi

B.E., University of Pune, India, 2008

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfilment
of the
Requirements for the Degree

Masters of Science

Athens, Georgia

2013

CacheMeter

by

Sagar Shailesh Sugandhi

Approved:

Major Professor: Kang Li

Committee: Lakshmish Ramaswamy
Tianming Liu

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2013

Acknowledgements

I would like to thank my major professor Dr. Kang Li for the feedback, encouragement and motivation that he gave throughout the course of this thesis. I greatly appreciate his help for improving this thesis while going through numerous drafts. I would also like to thank my committee members Dr. Lakshmish Ramaswamy and Dr. Tianming Liu for their valuable suggestions on the thesis. I also thank my parents and brother for always boosting my confidence. I would like to thank all my friends for their support and motivation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Internet is growing at a huge speed. According to ITU, 38.8% of the world population uses internet as of March 2013[2]. Web caching provides effective means for reducing bandwidth demands, improving web server availability, and reducing network latencies. HTTP is dominant protocol in the world wide web.

Caching specifications for HTTP/1.1 are specified in RFC 2616. The specification includes description of the HTTP/1.1 header fields related to caching, enumerates its values, and also provides an algorithm for determining whether a particular resource is cacheable, its lifetime, and its location of caching. This algorithm iterates on the HTTP response header field values.

In this thesis, we describe CacheMeter, our tool which calculates the percentage of HTTP(s) resources of a webpage(s) that are available for caching by the browsers. By using CacheMeter, we can calculate the cacheability statistics for one or more webpages. Currently, there is no tool which calculates such statistics.

CacheMeter is a 1200 lines code tool written in python 2.7. CacheMeter includes four phases which are described in details in chapter 3. CacheMeter employs RFC 2616 caching policy such that if none of the caching related headers or validators are present in HTTP

Response, we mark the resource as non-cacheable. If the headers are present, we determine the caching status by using RFC 2616 directions. CacheMeter only checks for resources which are received through HTTP GET Requests. We classify objects based on their content type and determine how many objects of each type are cached.

For testing CacheMeter, we ran it on Alexa[1] top 1000 websites. Alexa is a site that ranks websites based on traffic. We have used an approach which tries to maximize the number of webpages being tested on each website. Based on measurements computed by CacheMeter, we have a result section 4.3.

Results indicate that on an average, 77% of HTTP(s) resources per websites are marked as browser cacheable. The resource types which usually do not change frequently like Image, Javascript, CSS results to 85% browser cacheable. The cacheability percentage of only Images is 86%. For Javascript, CSS, and JSON, it is 80%. For Text objects, this statistic was 46%. 73% of HTTPS resources are browser cacheable. These statistics indicate the average percentage of resources per website, which are explicitly marked as browser-cacheable.

We also present caching statistics based on content size. Results based on content size indicate that larger sized objects have higher caching percentage. This meets expectation as larger objects have higher impact on network bandwidth.

Next, we describe our analysis for the HTTP caching implementation in Mozilla Firefox. Firefox employs an aggressive approach by caching all objects, even when response has no caching headers or has caching headers which do not allow caching of Response. It uses the RFC specified policies while using the cached entries, by checking the Headers of the cached entry. It tries to make subsequent requests conditional by asking the server to validate the cache entry, in case the cache entry does not have caching headers or is expired.

We present our results for comparison of various Firefox releases with respect to implementation of HTTP caching. We start with Firefox v3.6 and end with Firefox v21.0 which was the rolled of version when the experiments were conducted. We analysed few conditions related to caching on Firefox; for example if the received HTTP response is broken, Firefox eliminates it from the cache. Also, if a HTTP transaction is cancelled, but the received response is complete and cacheable, then Firefox caches it.

Finally we conclude with our results for CacheMeter and Firefox HTTP caching behaviour.

Chapter 2

Related Work

Caching is important aspect of the web. Caching has several advantages which includes reduction in access latency, reduction in network bandwidth consumption, and reduction in server load. Caching can be done on client side, on network proxies, and on servers. But if the cache does not provide a fresh copy to the client and instead provides a stale version, then value of caching significantly reduces. Hence maintaining cache consistency is a crucial aspect.

Barish & Obraczka[4] discuss various caching designs and their implementation. Caching designs they discuss include Proxy Caching, Reverse Proxy Caching, Transparent Caching, Adaptive Web Caching, Push Caching, and Active Caching. Proxy Cache is present between the client and the origin server. It gets the request from the client. If it has the object stored, it returns from the cache, otherwise it sends request to the origin server. After getting the response, it possibly saves the response and forwards it to the client. Reverse proxy caching involves deploying the cache near the origin server which allows the origin server to achieve high quality of service. Transparent Caching is similar to Proxy caching but instead of getting the request from the client, it intercepts it. Adaptive caching on the other hand is a technique which is based on demand of the content such that multiple cache join a group

if the demand for the content increases. Push caching is a concept which states that the data be moved geographically closer to the location where majority of the request originate. Active caching involves cache applets which compute the dynamic user-personalized data locally without requiring a post back to the server.

Gwertzman & Seltzer[5] talk about various cache consistency techniques. This mechanisms are used to update the cache entries so that it has fresh records. Cache consistency mechanisms that are used in the internet are time-to-live fields, client polling and invalidation protocols.

This website[7] discusses about various caching related bugs in Microsoft Internet Explorer, Netscape Navigator, and Opera.

Mozilla Firefox gives various preferences to the user with respect to browser caching. This preferences include:

1. network.http.use-cache:

   This is of type boolean. If this is true, browser checks the cache before sending any request for a resource.

2. browser.cache.check_doc_frequency:

   This preference controls how often a newer version of a cached resource be checked. The four values allowed are Always, Never, Once every session, and whenever the resource expires.

3. browser.cache.disk.capacity

   browser.cache.disk.enable

   browser.cache.disk.max_entry_size

   These 3 preferences are for setting the maximum capacity of the disk cache, whether cached resources should be saved on disk, and the maximum size of a single resource that is allowed to store on disk

4. browser.cache.disk.disk_cache_ssl

   This preference determines whether a resource transmitted over SSL should be cached on disk.

5. browser.cache.memory.enable

   browser.cache.memory.max_entry_size

   These 2 preferences control the in-memory caching of objects

Chapter 3

Approach

3.1 Overview of CacheMeter

For each website name taken from the list of websites, a set of unique URL is grabbed and saved as link-set for that particular website. The links from the link-set are then individually run inside Firefox to generate a collection of HTTP GET Requests for various HTTP(s) objects on that web page. We then separately send this HTTP requests from code to grab its Response Headers. The Response Headers are parsed by the Analysis component to determine whether individual resources are cacheable.

The results are stored in csv format, text format. Text format makes it easy to search for the results of any one particular website. Text format contains detailed results. csv format is more concise representation of the statistics and used for creating graphs. Graphs are generated to visually depict the results.

Figure 3.1 gives the overview of CacheMeter. CacheMeter is written in python 2.7.

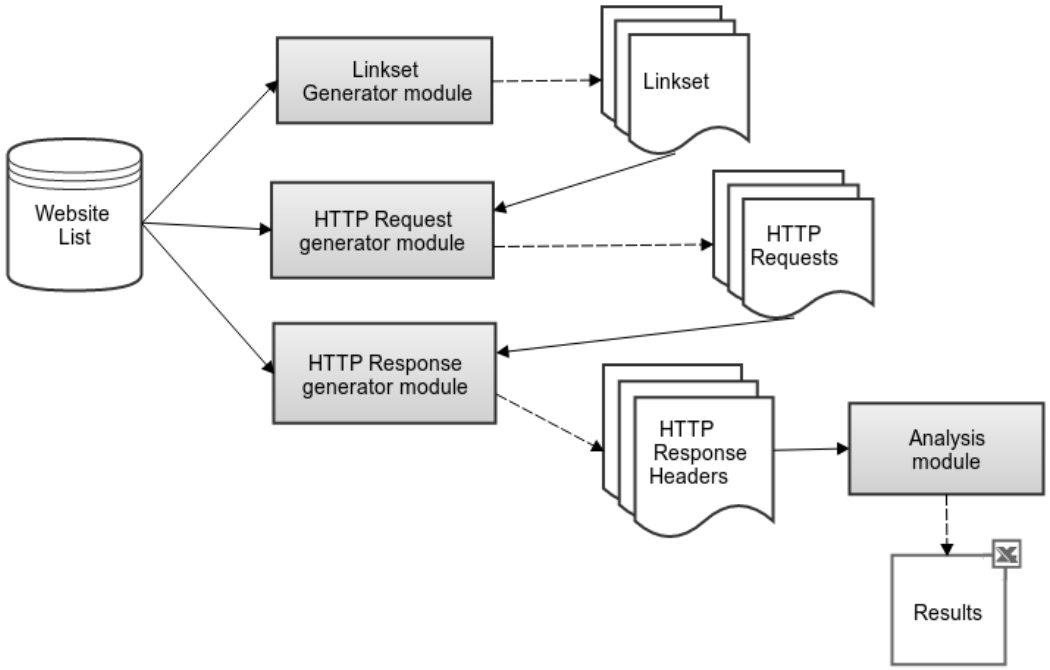

Figure 3.1: System Architecture - CacheMeter

3.2 Implementation

Each phase of CacheMeter is described below:

3.2.1 Phase 1: Creating Link Set

For each website in the dataset, we generate a set of links. This is achieved by a web crawler which runs on each website and selects links with the same domain name as the selected website. The web crawler is a modification of [12]. It checks for URL's which contain the same TLD and Second level domain. Such URL's are added to the list. This phase gives us a variety of links associated with a particular website.

The Link Set for each website is stored in *data/linkset/* folder with the name of the file as $< website - name > .linkset$. Figure 3.2 summarizes this phase.

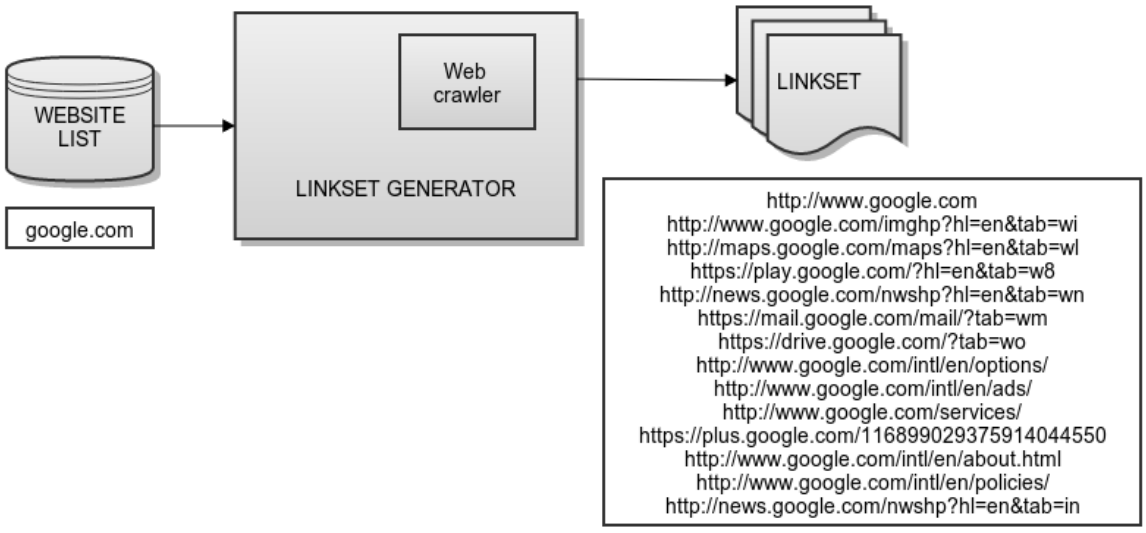

Figure 3.2: Link Set generator

On an average, we have 21 separate URL's per site after executing this module. For few sites, the Link set contains a very small number. For live.com, link set generator emits only 1 link. There are 196 other sites which have only 1 link.

3.2.2 Phase 2: Generating GET Requests from a browser

At the end of Phase 1, we have a set of URL's to work on. For each website in the dataset, we look for its Link Set and open each URL from the Link Set on Firefox. We have used Selenium WebDriver's PythonBindings [11]. This helps in the automated opening of all the URL's inside Firefox. We have added an extension to Firefox named HTTP Request Logger [10]. HTTP Request Logger logs the request generated from Firefox and stores them in a text file in the following format:

```
<site-referrer>  REQUEST-METHOD  <object-URI>
```

This module grabs HTTP Request URI for all the resources on the web page loaded in Firefox. It then filters out URI's grabbed through GET Request method. Repeating Requests for the same resource are also filtered out.

These are then stored in the folder $data/requests/$ with file name as $< website-name >$ $.request$. Figure 3.3 summarizes this phase.

Our approach includes a web browser to generate HTTP Requests in order to cover all the objects that are loaded for a particular link. The adjunct HTTP(s) Requests generated for all the images, iFrame contents, JavaScripts, CSS, videos, etc. that are required to load a webpage are logged by opening the page in Firefox.

A downside to this approach is that the HTTP Request collection will also include objects from a different website. For example, in case of an advertisement, resources might be from a different website than the one being processed. We would like to eliminate such objects from the Request set, but we do not have a solution in this thesis. We argue that the number of such objects would be small as compared to the total number of objects processed for the website. Hence it will only have a minimal impact on the calculated statistics.

We only consider HTTP(s) GET Requests. Response received from POST Request are not cacheable [3].

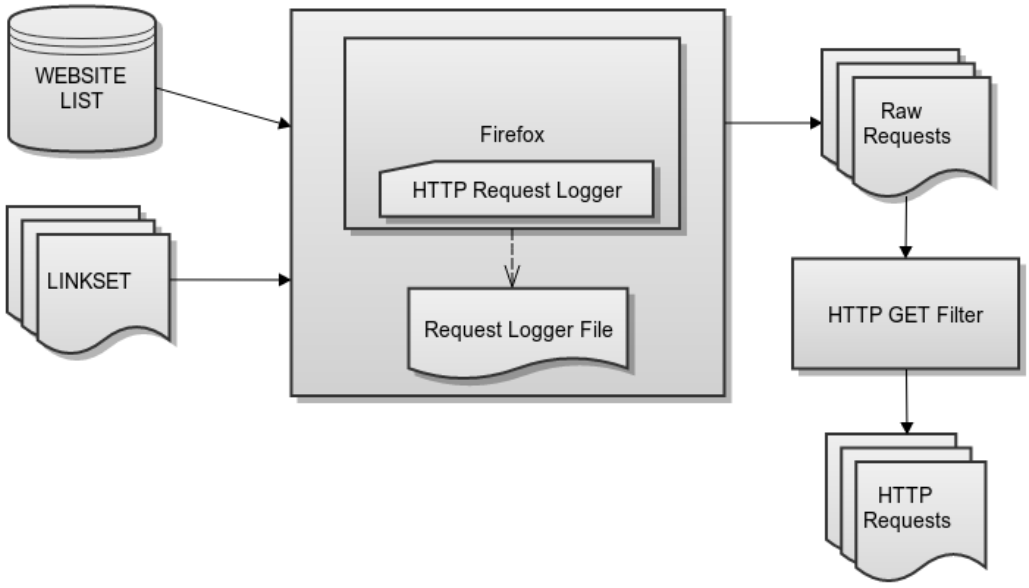

Figure 3.3: HTTP Request Generator

3.2.3 Phase 3: Getting HTTP response headers associated with caching

Phase 3 and Phase 4 constitute the components of CacheMeter. For each website in the dataset, this component searches for the associated HTTP request file and sends GET request to each resource. We have used the urllib2 package for python for sending requests. We send GET requests instead of HEAD request as we are also interested in the Response data size along with the Response headers. After getting the HTTP response headers, we only save the HTTP response header fields associated with caching. While invoking HTTP request through the code, we do not send any HTTP request headers associated with caching policy. Hence we do not influence the caching policies set by the website. The response files are stored in the $/data/responses$ folder with file names as $< website - name > .response$

Figure 3.4 summarizes HTTP response generator and Analysis component.

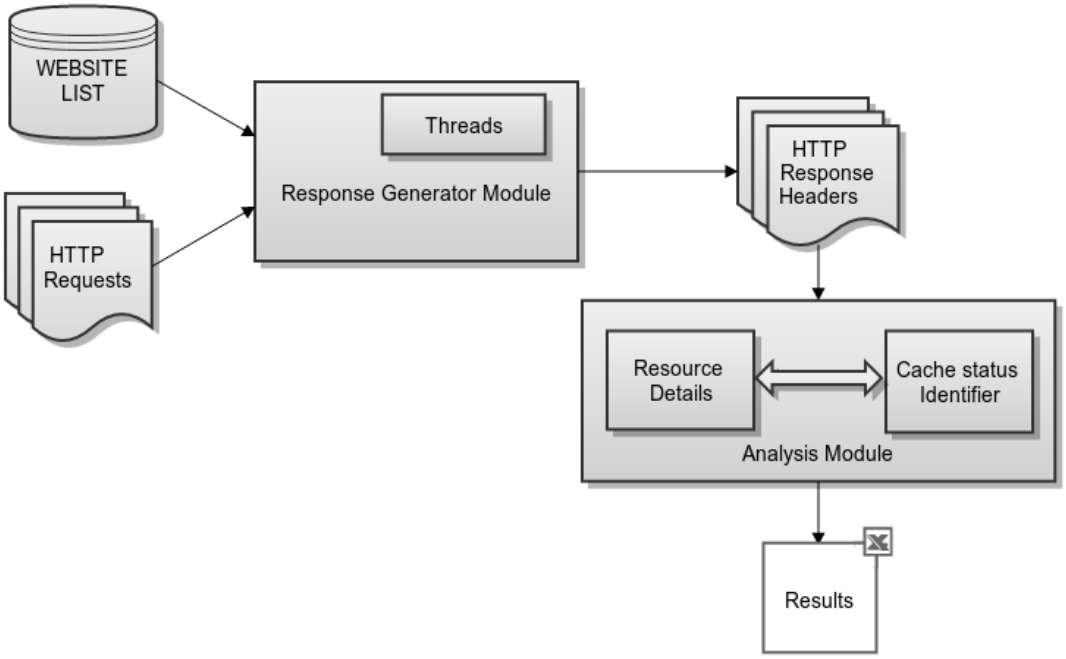

Figure 3.4: HTTP Response generator and Analysis component

3.2.4 Phase 4: Saving HTTP response and calculating caching percentage

For each HTTP resource, we save the following HTTP Response header fields for processing:

1. Content type,
2. Content size
3. Age,
4. Expires,
5. Pragma (HTTP/1.0 - deprecated), and
6. Cache-control response directives

According to RFC 2616, Cache control response directives points to the caching policy forced by the server. [3]. Its values that we are interested in are public, private, no-cache, no-store, and max-age.

According to RFC: By default, a response is cacheable if the requirements of the request method, request header fields, and the response status indicate that it is cacheable.

Based on the values for the Cache control response directives field, we use the below algorithm to determine whether the resource is cacheable or not. We employ an approach such that when cache-control directives, Expiry response header or validators are not present in the Response Headers, we mark it as non-cacheable.

1. We check if Response status code is one of the following 200, 203, 206, 300, 301, 410. Only then we consider the resource and move to step 4.

2. If response status code is either 302 or 307, then we check if any Cache control directive or expiry are set in the HTTP response. If these directives are set, only then we consider the resource by going to step 4.

3. We skip resources with other status codes and continue with next resource.

4. We check if Last-Modified Header is present in the Response Header. If so, we check if Cache-Control=no-cache is present in the Response Header. If not, then we set validator = true for that resource.

5. If the Cache-control Response-directives contains public or private, we mark the resources as may-be-cacheable. We then goto step 6.

6. If Cache-control Response-directives contains no-cache or no-store, the resource is considered non-cacheable. We flag the resource as non-cacheable and goto step 10.

7. If the Pragma=no-cache (deprecated) directive is present, then we flag the resource as non-cacheable and and goto step 10.

8. If max-age directive is present in the resources response headers and if its value is greater than 0, then the resource is cacheable. We flag the resource as cacheable and goto step 11. No-store and no-cache directives override max-age.

9. If Expires response header is present, we check if its valid. If it is less than the current date, we mark the resource as non-cacheable and goto step 10. If it is greater than current date, we mark the resource as cacheable and goto step 11. Max-age overrides Expires.

10. If resource is non cacheable but validator=true for the current resource, we mark the resource as cacheable.

11. If Vary=* is present, then we consider the resource as non-cacheable.

12. We consider all other resources to be non cacheable.

3.2.5 Type of resources

We classify resources into based on their Content-type header in the HTTP response. Table 3.1 gives details about how we classify the resources based on content-type.

Icon	containing "icon" like image/x-icon
Image	all other types starting with "image" like image/png
CSS	text/css
JavaScript	containing "javascript" like text/javascript and application/javascript
Json	application/json
Text	all other types starting with "text" like text/plain, text/html
Video	types starting with "video" like video/x-flv
Application	all other types containing "application"

Table 3.1: Types of resources based on content-type

Furthermore, to depict the results, we combine CSS, Javascript, and Json categories into the "Scripts" group. The other group is "Image" group which consists of Image, and Icon. The third group is "Text" which only consists of Text category

We store the number of each resource type and count of how many of them are marked as cacheable along with the total caching percentage for all websites from our dataset in $data/results/$ folder with the file name as $< date > .txt$. We also save the size of each object.

We create another .csv file which contains comma separated value version of above information for the purpose of plotting graphs.

Chapter 4

Experiments and Results

4.1 Data set

CacheMeter was run on 967 websites taken from Alexa [1] top 1000. Alexa ranks the sites based on their traffic statistics. Alexa has top 1 Million site ranks available for free download. The list includes sites from around the world. This dataset is updated daily by Alexa. We download this dataset on the day of experiment and use filtered version of top 1000 sites. This list is stored in a text file and is input to CacheMeter.

4.2 System details

The experiment were run on machine with Intel Core i5-2410M CPU @ 2.30 GHz processor. The system has RAM of 4 GB. It has 25 GB hard-drive. The wireless card is Intel Corporation Centrino Wireless-N 1000. The Operating system installed is 32-bit Ubuntu 12.04 LTS.

4.3 Results

The results were calculated across 967 websites with an average number of resources checked for a single website being 501. CacheMeter takes approximately 96 hours to run on 967 websites with an average of 6 min. per website. The results shown in this document is for a run started on June 1, 2013.

Sample Linkset for youtube.com is shown below:

```
http://www.youtube.com/
http://www.youtube.com/upload
http://www.youtube.com/music
http://www.youtube.com/sports
http://www.youtube.com/gaming
http://www.youtube.com/movies
http://www.youtube.com/YouTubeShowsUS
http://www.youtube.com/news
http://www.youtube.com/channel/UCBR8-60-B28hp2BmDPdntcQ
http://www.youtube.com/feed/UCry7B7DGVgUIa6k4Tis_DJQ
http://www.youtube.com/feed/UCBR8-60-B28hp2BmDPdntcQ
http://www.youtube.com/feed/UCHF66aWLOxBW4l6VkSrS3cQ
http://www.youtube.com/feed/UCK7tptUDHh-RYDsdxO1-5QQ
http://www.youtube.com/feed/UCf4FYTsGFFcdc68AUPIU3RA
http://www.youtube.com/channels
http://www.youtube.com/watch?v=XMF22_MEMJU
http://www.youtube.com/channel/UCDsZrtnDOLWvOPY8lmYKoxA
http://www.youtube.com/channel/HC8x43Y3JC3mo?feature=g-logo
http://www.youtube.com/watch?v=0LP3Zs_V_BQ
http://www.youtube.com/user/WashingtonFreeBeacon
http://www.youtube.com/channel/HCAqPF7g4Dywo?feature=g-logo
http://www.youtube.com/watch?v=NlOF03DUoWc
http://www.youtube.com/user/WarnerBrosPictures
http://www.youtube.com/channel/HCUnGt6FaatzE?feature=g-logo
http://www.youtube.com/watch?v=KbWgUO-Rqcw
http://www.youtube.com/user/Darkbeatdk
http://www.youtube.com/channel/HCQVDIyIy_jKQ?feature=g-logo
http://www.youtube.com/channel/HCxAJ-ON2kZuw?feature=g-logo
http://www.youtube.com/watch?v=VKCvrPew9I8
```

A sample Request and formatted Response with meta data for a resource on youtube.com is included below:

```
http://i2.ytimg.com/li/UnGt6FaatzE/hqdefault.jpg

URL           : http://i2.ytimg.com/li/UnGt6FaatzE/hqdefault.jpg
Data length   : 21193
Status Code   : 200
Content type  : image/jpeg
Last modified : Mon, 01 Jul 2013 18:14:46 GMT
Expires       : Tue, 02 Jul 2013 00:14:46 GMT
Cache control : public  max-age=21600
```

The above resource of type Image is marked as cacheable as Cache control allows it to be stored in browser cache and also specifies a freshness lifetime. It has Last modified directive which is used for sending conditional request if the Response is stale on subsequent request.

We have stored the results in the text document. Following is a snapshot of result for youtube.com, and amazon.com

```
youtube.com - 2013-07-01 14:13:20

Record Type          Total      Cached     Non-Cached

text                 109        23         86
image                506        466        40
javascript           30         25         5
application          14         11         3
json                 1          0          1
video                9          0          9
css                  9          8          1
icon                 3          2          1
                     -------    -------    -------
                     681        535        146

Percent Cached: 78%
Objects that fall in cacheable category (Images, Scripts, Application): 563
```

```
HTTPS Requests: 50
HTTPS Cached Percent: 90
----------------------------------------

amazon.com - 2013-07-01 14:13:20

Record Type          Total      Cached     Non-Cached

text                 177        7          170
image                694        619        75
javascript           78         61         17
application          8          7          1
json                 3          2          1
css                  39         38         1
icon                 2          1          1
                     -------    -------    -------
                     1001       735        266

Percent Cached: 73%
Objects that fall in cacheable category (Images, Scripts, Application): 824
HTTPS Requests: 53
HTTPS Cached Percent: 83
```

The results show that 681 objects were processed for youtube.com out of which 535 objects were marked for browser-caching by the website. Hence, 78% of the objects are browser cacheable for youtube. The result also shows the various object types encountered and their count. It also indicates the caching count of each type. 50 out of 681 objects are result of HTTPS request. The results show that 90% of the HTTPS objects are browser-cacheable.

For the 9 videos that are not cached on youtube.com, the Expires date is set to a date in past. According to RFC 2616, this is done by the websites to ensure that the cached entry is validated before reuse. But since the Response does not have Last-modified header (the validator), CacheMeter marks this resource as non-cacheable. On a subsequent request, this response will not be used from cache as it will be stale and their is no validator information present to validate the cached response from server.

Graph 4.1 shows size wise distribution of various objects on youtube.com.

Number of links in linkset can be configured. On an average, we have 21 links per website. Our results for youtube was achieved on 30 URL's in the linkset. Here we show result for youtube.com with 1 URL and 15 URL's respectively in the linkset.

```
youtube.com - 2013-07-03 09:54:40  {1 URL in LINK SET}

Record Type          Total      Cached     Non-Cached

text                 8          1          7
image                33         31         2
javascript           9          8          1
application          3          2          1
css                  3          2          1
icon                 1          0          1
                     -------    -------    -------
                     57         44         13

Percent Cached: 77%
Objects that fall in cacheable category (Images, Scripts, Application): 49
HTTPS Requests: 1
HTTPS Cached Percent: 0
----------------------------------------
youtube.com - 2013-07-03 10:01:46 {15 URL's in LINK SET}

Record Type          Total      Cached     Non-Cached

text                 76         25         51
image                446        422        24
javascript           14         13         1
application          6          5          1
json                 1          0          1
video                8          0          8
css                  5          4          1
icon                 2          1          1
                     -------    -------    -------
                     558        470        88

Percent Cached: 84%
```

```
Objects that fall in cacheable category (Images, Scripts, Application): 474
HTTPS Requests: 62
HTTPS Cached Percent: 95
---------------------------------------
youtube.com - 2013-07-03 12:41:02 {30 URL's in LINK SET}

Record Type          Total      Cached     Non-Cached

text                 142        51         91
image                603        562        41
javascript           27         23         4
application          12         11         1
json                 1          0          1
video                15         0          15
css                  7          6          1
icon                 2          1          1
                     -------    -------    -------
                     809        654        155

Percent Cached: 80%
Objects that fall in cacheable category (Images, Scripts, Application): 652
HTTPS Requests: 71
HTTPS Cached Percent: 95
```

The range of caching percentage is in 77% to 84% range when the number of url's in the link set is tuned.

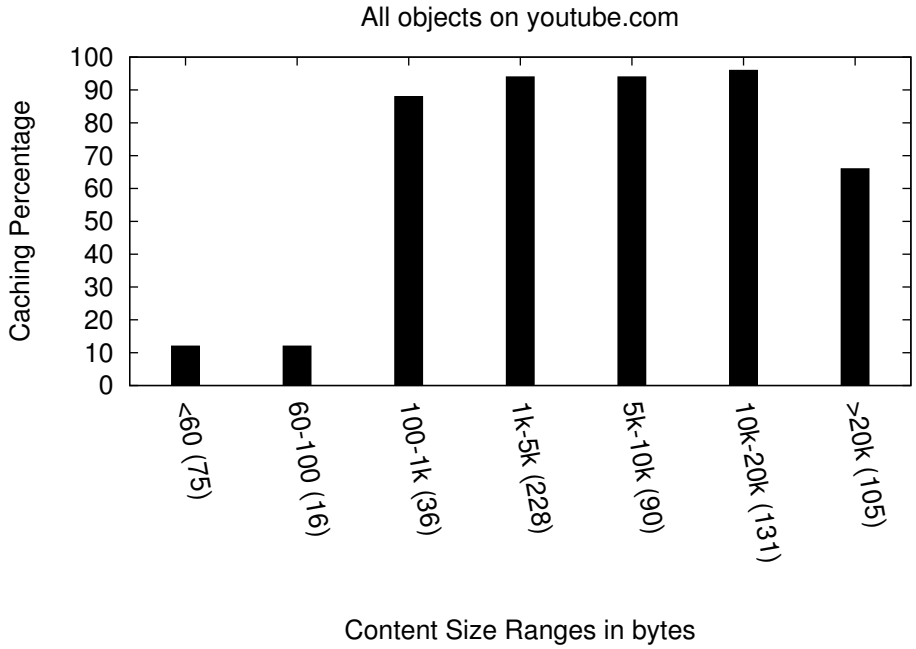

Figure 4.1: Size wise distribution for youtube.com

For amazon.com, 1001 objects were computed off which 735 are available for reuse from the browser cache. 53 object were obtained by HTTPS requests. 83% of those were available for caching by browsers.

Graph 4.2 shows size wise distribution of various objects on amazon.com [1]

It should be noted that the number of resources are different for each website. We have kept an upper bound of 1000 on the number of resources. The minimum number of resources encountered is 1 and the average is 501.

We define λ as the minimum number of resources that should be present in the request set of a website. For $\lambda = 50$, number of websites is 864. Thus this removes 103 sites from consideration due to number of HTTP request in the request set being less than 50.

Mean calculated across 864 websites show that 77% of HTTP(s) resources per website are marked as cacheable. 519 out of 864 websites have caching percentage of more than 77%.

[1] Content-size for 1 resource on amazon.com was not calculated by CacheMeter

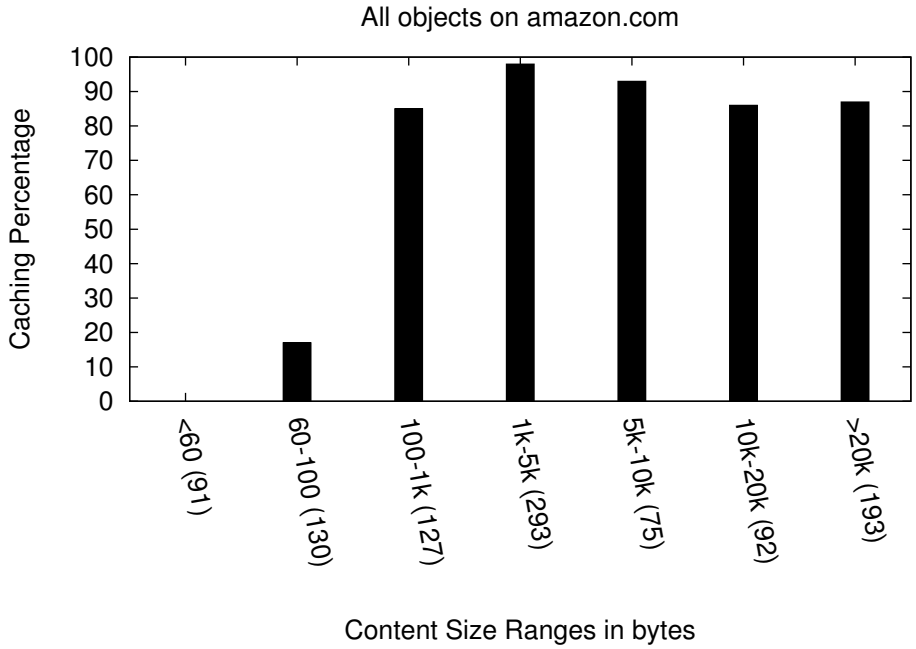

Figure 4.2: Size wise distribution for amazon.com

85% of infrequently changing objects are cacheable. 73% of HTTPS resources are cacheable.

Graph 4.3 plots the ratio of websites (y-axis) that fall within a given caching percentage (x-axis). We plot this data for All objects, Images, Text, and Scripts (JavaScript, json, and CSS content types). The graph shows that for text objects, the number of websites that have caching percentage less than 50% is more than 0.5. For images, the ratio of websites that fall below 60% caching is less than 0.1, and from then it increases sharply as nearly half of the websites have Image cacheability of 90% or less and the remaining half having cacheability percentage of more than 90%. Scripts content types also show higher cacheability percentage, with more than 40% websites having cacheability percentage of above 90%.

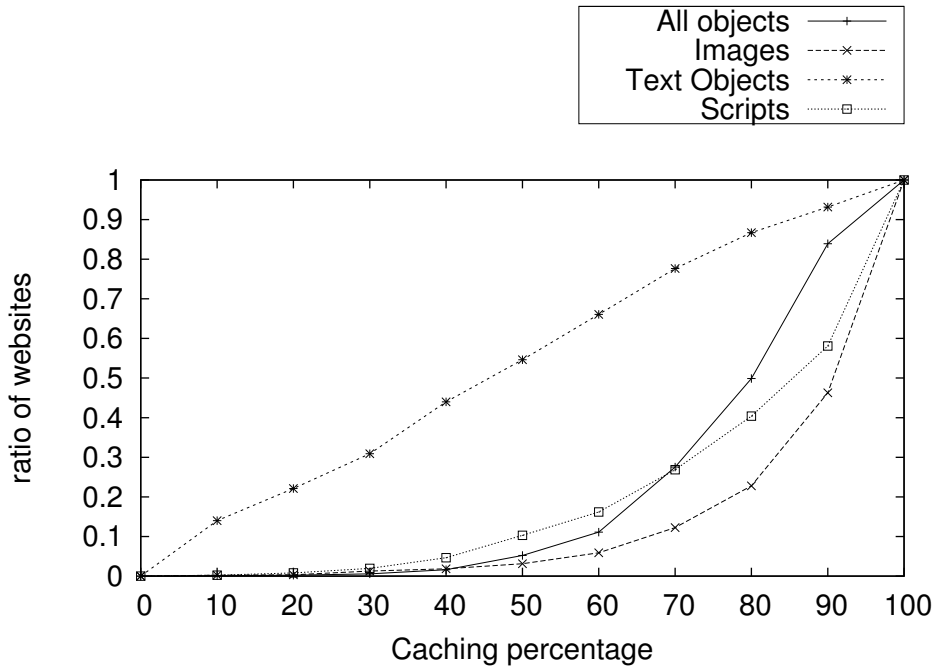

Figure 4.3: Caching Statistics for static resources

Graph 4.4 shows size wise distribution. x-axis shows the caching percentage based on size of objects. The formula used is

$$\frac{\text{size of objects cached}}{\text{size of total objects}} \times 100$$

We calculate this percentage for each website and plot Graph 4.4 which shows the ratio of websites that fall below certain caching percentage(x-axis)

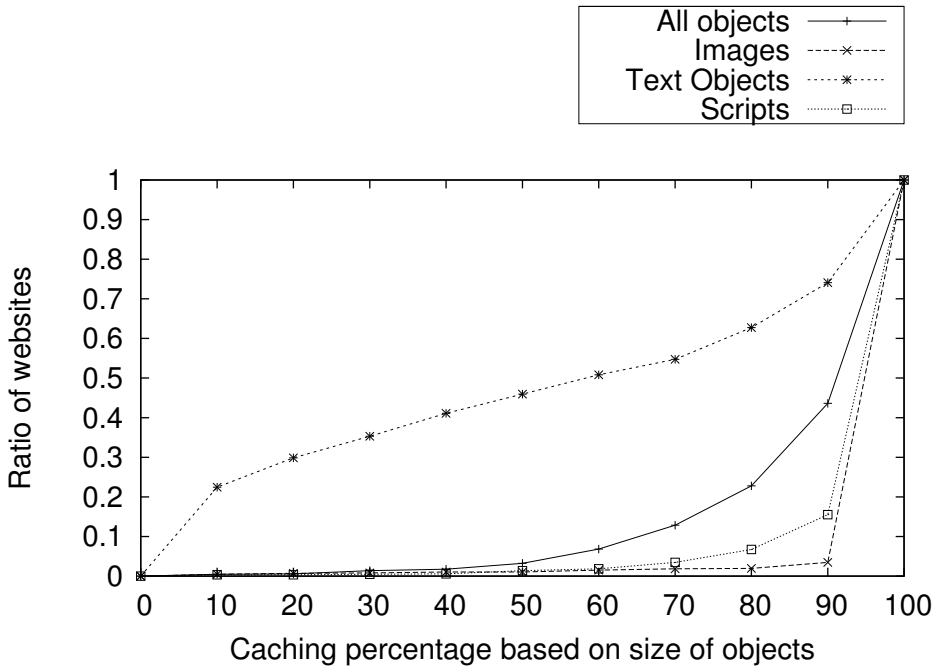

Figure 4.4: Caching Statistics for static resources

Graphs 4.5, 4.6, 4.7, and 4.8 show additional statistics based on size of objects. We have divided the results in range of sizes. These graphs show the average cacheability percentage of size of objects encountered in each range. Graph 4.5 shows the caching percentage distribution of all objects encountered across 967 websites.

For Image group, the cacheability percentage is 53% for content-length less than 100 Bytes. It is around 98% for size grater than 100 Bytes. For Text group, the cacheability is consistently low for all sizes with objects with size more than 20 KB having highest cacheability percentage of around 61%. Even for Script resource group, the cacheability percentage is low for smaller sized objects and increases with the increase in object size. For size range greater than 20 KB, the caching percentage is highest and is nearly 94%. Thus objects with larger size have higher caching percentage. This meets expectations as larger objects have higher impact on network bandwidth, and eventually on user perceived latency.

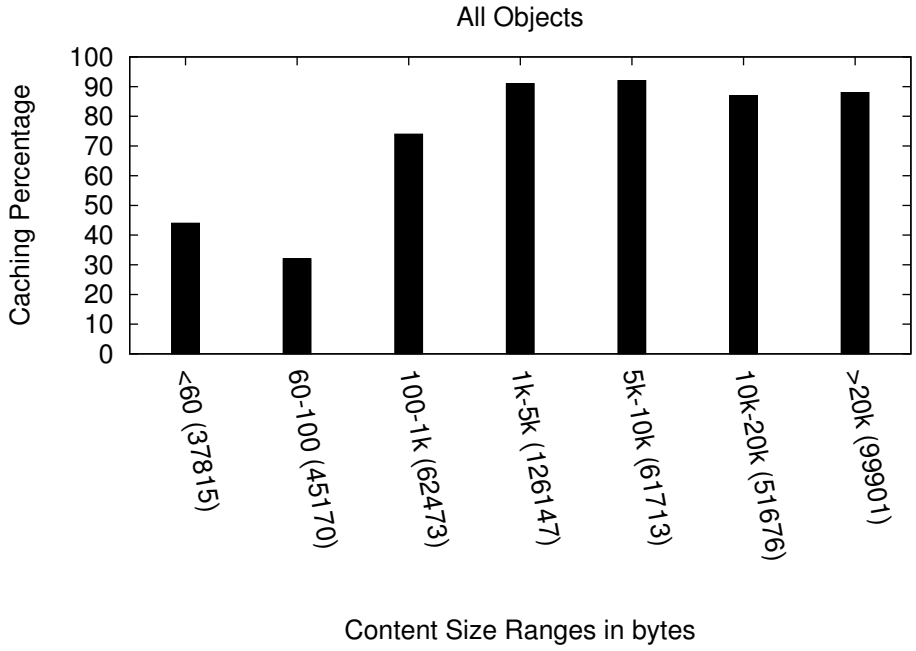

Figure 4.5: Caching Statistics for All objects - Content Size

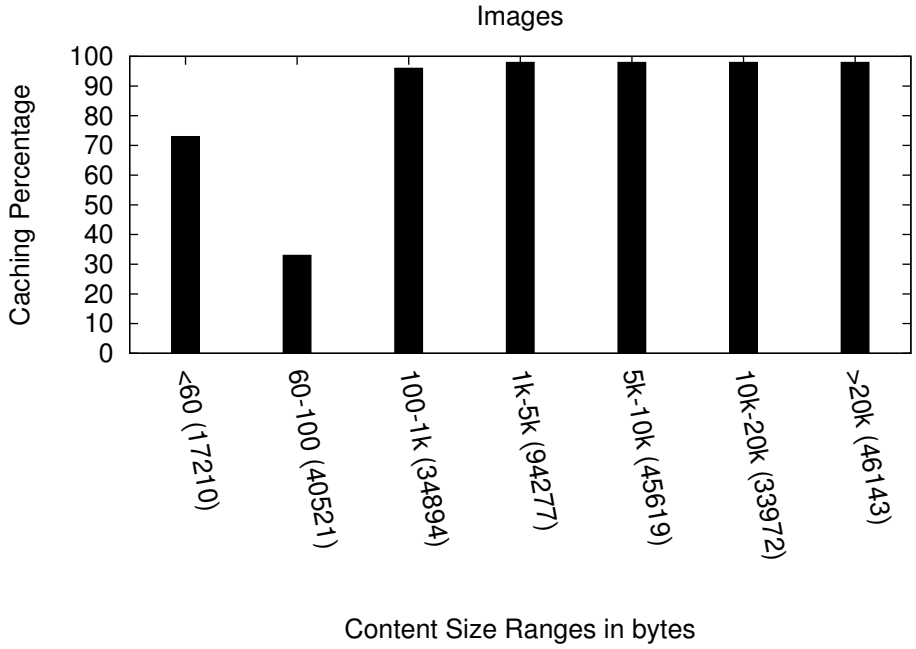

Figure 4.6: Caching Statistics for Images - Content Size

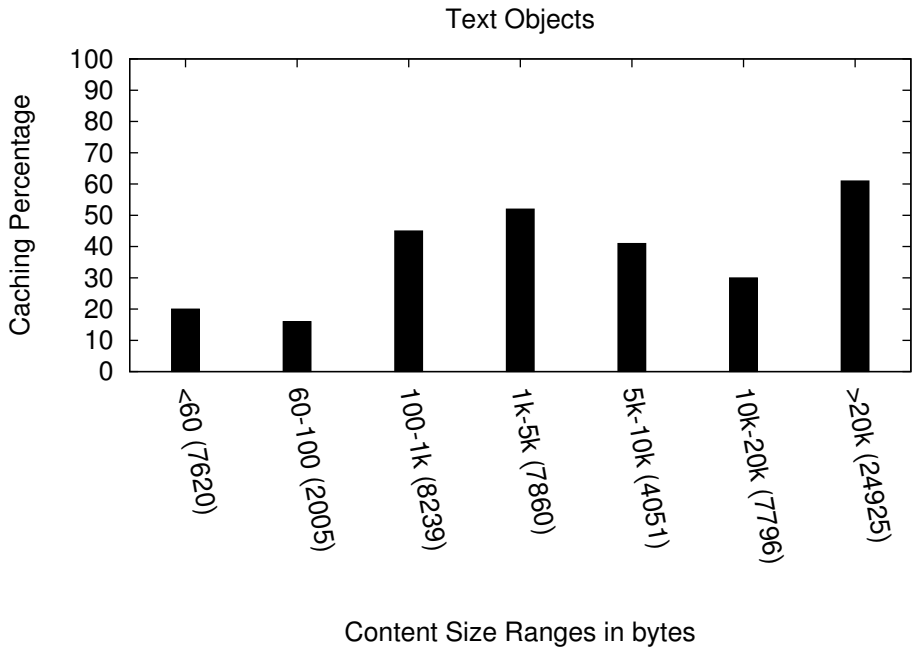

Figure 4.7: Caching Statistics for Text - Content Size

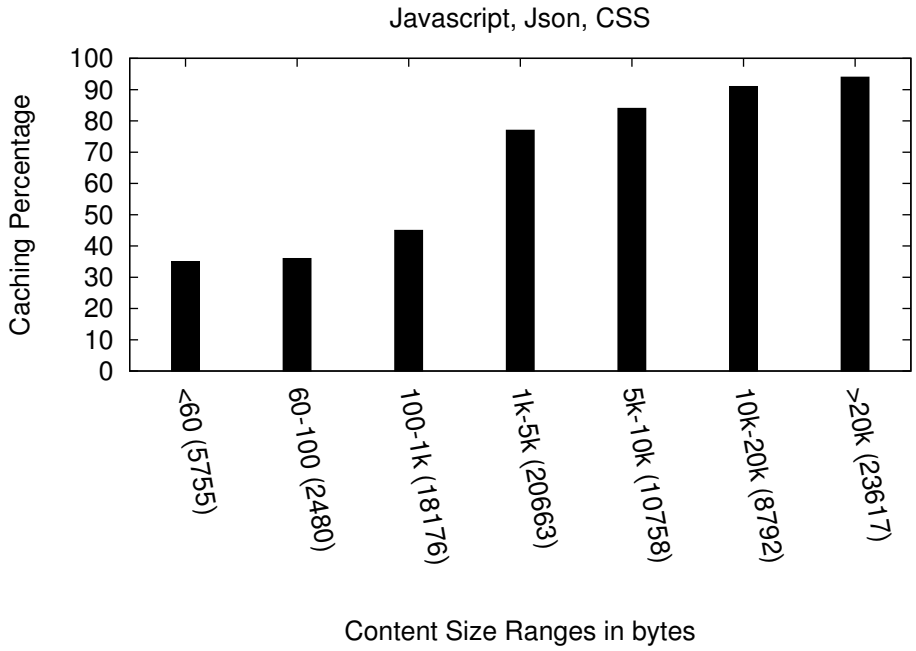

Figure 4.8: Caching Statistics for Script - Content Size

Graph 4.9 gives comparison statistics of two different measurements. We have used the Cache Header with validator to determine the caching statistics. Cache Header without validator is an approach where we just check if the response is marked for Caching by checking the Expires headers and the max-age headers. We do not consider Last-modified Header in this check.

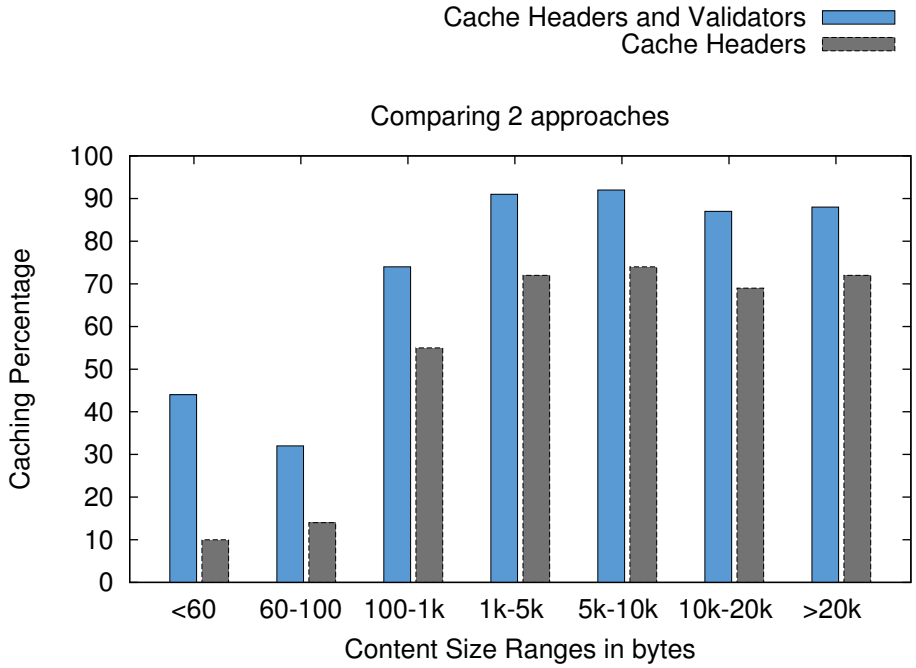

Figure 4.9: Caching Statistics for two approach - Content Size

Chapter 5

HTTP caching in Firefox

5.1 Method of checking

We studied Mozilla Firefox code for HTTP caching through Mozilla Cross Reference [13]. We verified our observation by using Mozilla debugging tool for HTTP_logging. It logs function traces in HTTP flow along with the HTTP Request and Response that is generated.

- We used the following commands as specified in [17] for enabling the logging

```
export NSPR_LOG_MODULES=nsHttp:5
export NSPR_LOG_FILE=log-http-flow.txt
```

- We clear the Firefox cache, and open a website: google.com. We store the generated HTTP log.
- We restart the browser. Open google.com again. We store the generated HTTP log.
- By looking at the logs, we identify the flow for objects that are loaded from cache and objects for which a HTTP request is sent.

5.2 Firefox Implementation

Before connecting to the network, there is a check whether "use-cache" preference is set to true by the user. If it is set then OpenCacheEntry is called. It checks whether the request method is POST. if so it appends an unique post id to the cache key. If the request method is other than POST, GET or HEAD, then cache is not used and the method returns. If request method is one of these 3, then Cache key is generated, Cache access mode and storage policy is determined. HttpCacheQuery is formed based on above parameters, and the query is executed. If cache entry is available then HttpCacheQuery::OnCacheEntryAvailable callback is called, which then calls CheckCache method. CheckCache determines whether the cache entry is valid.

It checks if the cache entry was opened for reading. If false, the method returns. It then gets the method that was used to generate the cached response. If the request-method is HEAD, then it continues to check cache entry only if the current request is also HEAD, otherwise it returns.

If the Status code in the cached response header is 304, then do not return it from cache, and abort [14]

If the cached content-length is set and it does not match the data size of the cached content, then the cached response is partial. so it either issues a byte range request or refetch the entire document based on whether the byte range request was successfully created.

From here on, the cache entry's Response headers are checked to determine if it can be used to fulfil the current HTTP(s) Request:

1. Check the vary response headers to determine if the Response would vary. if true, set validation flag to true

2. If $check\_doc\_frequency$ user preference is set to 1 which means never load from cache, then set validation flag as true

3. If *check_doc_frequency* preference is set to 2 which means always load from cache then set validation flag to false unless if no-store header is present or if using SSL with no-cache response header, in which case set validation flag to true.

4. Check for no-cache, no-store directives in the cached response header. If present set validation flag as true. Also check status code received in response. If 303, 305 or 4XX, then set validation flag as true.

5. If query string is present in the GET request and expires and max-age headers are not explicitly set in the query string, then set validation flag as true

6. check if the cache entry has expired by calculating the expiration time. If current time is greater than expiration time, then set validation flag as true; else if must-revalidate header is present in the response, set validation flag as true

7. if validation flag is still unset and User has defined If-Match header, if the cached entry is not matching the provided header value or the cached ETag is weak, force validation by setting the validation flag

If validation flag is set at this moment, then HTTP request will definitely be issued to the server. It then tries to make the request conditional without overwriting user-defined conditional headers. It adds If-Modified-Since header if a Last-Modified is present in cached response header and validation is forced based on the Vary response header.

nsHttpChannel::OnCacheEntryAvailable callback gets called which then eventually calls ContinueConnect. ContinueConnect checks whether we have a valid cache entry (determined in CheckCache), and if so ReadFromCache is called. Else network is hit.

ReadFromCache actually processes the cached response that was checked in CheckCache.

If cache entry is not valid and Network is hit with the request, then the cache listener is installed on the transaction listener and the generated response is stored in cache as it

arrives in through the http channel. Once the data is completely arrived, the cached content is finalized. For this process OnStopRequest, and CloseCacheEntry methods are important. OnStopRequest is called when all the data has arrived.

- If the content received is partial, the transaction completed successfully and the transaction is from the network, then it is an error as byte-range transaction finished prematurely.
- If the content received is partial, the transaction completed successfully and the transaction is from the cache, try to check if there is any data in the transaction pipeline. if true, return from this method
- If the transaction failed, cancel transaction

OnStopRequest dooms the cache entry if the transaction is aborted. This is achieved by passing boolean false to CloseCacheEntry. So, If the content received is not complete, the cache entry will be doomed. This can happen if the network connection fails amidst the transaction. If the data is loaded successfully, but the transaction is cancelled, the content is still saved in cache for future consideration.

Figure 5.1 summarizes the caching flow.

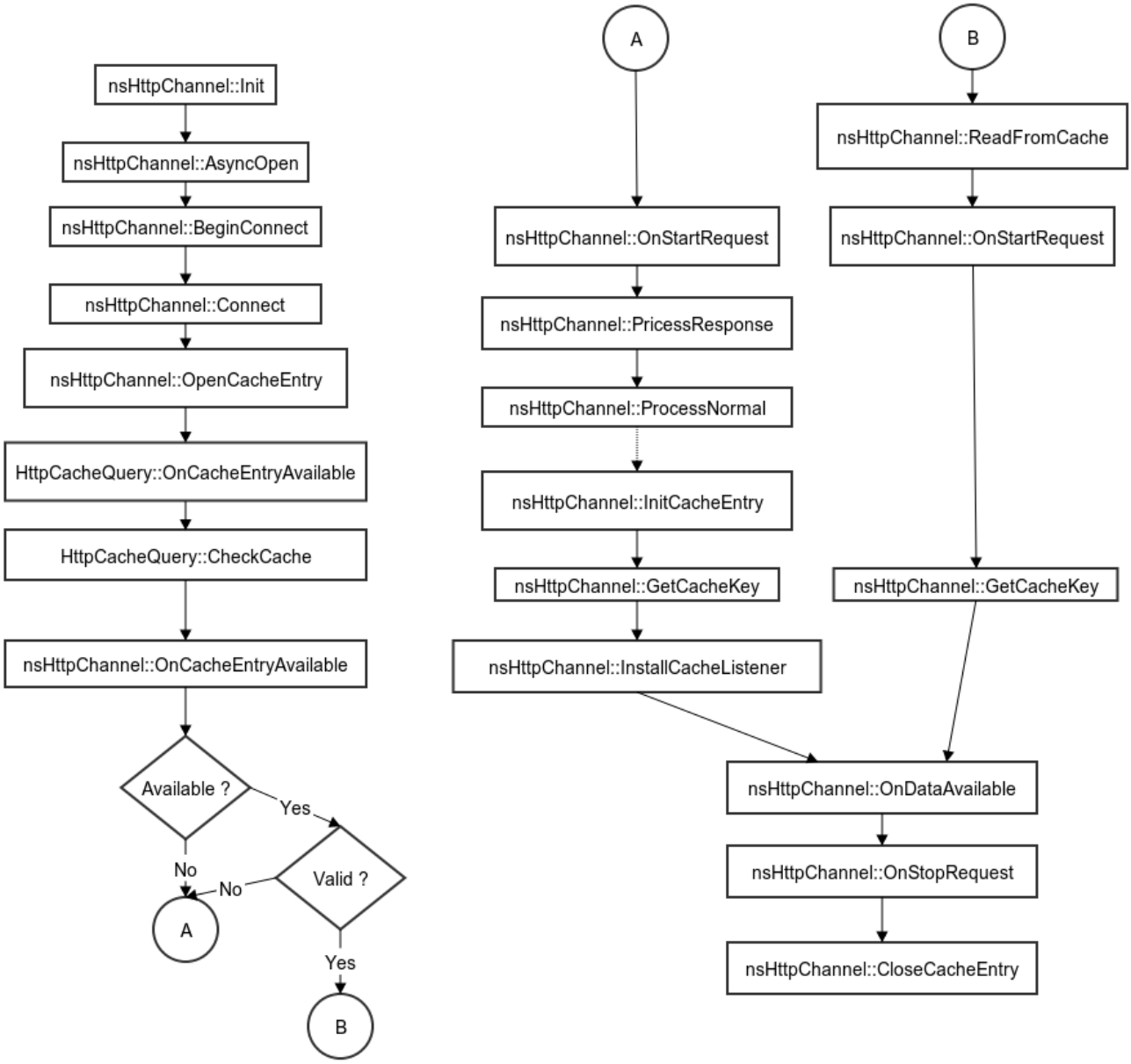

Figure 5.1: Firefox HTTP caching flow

5.3 Summary of behaviour

We found that Firefox behaves differently in two scenarios; when no-store is present in Cache-control response directive, it will still cache the response. Also when no caching

related headers and validators are present in the response, still Firefox caches it. This is not what is expected by RFC.

Firefox employs an aggressive approach as it caches all the HTTP Responses that have been completely received. Before sending a subsequent request, if a cache entry is available, Firefox uses various validation mechanisms which include checking the cache headers, checking the freshness from max-age or Expires header, and checking user preferences. This validation mechanisms specified by Firefox follow the RFC policies.

It tries to send conditional request to the server if the response has expired or no-cache cache directive is present, and if the Last-modified header. If the resource is not modified on the server from the specified Last-modified date, then the server will send a 304 Not modified Status code response and hence Firefox can use the cached version. If the resource is modified on server, the server sends the fresh response.

If the HTTP transaction is cancelled but the HTTP response received is complete, then Firefox will cache the response.

5.4 Handling of broken resources

Firefox does not cache any HTTP Response which is broken. On the other hand, it caches every HTTP Response that is complete even if the transaction is cancelled.

Note that by broken resource, we mean the resources which were not intended to be partial. Firefox caching has a different policy for handling partial responses (status code 206).

For verifying the behaviour in case of broken resources, we created a web-page with XML HTTP Request(XHR). We cancel transaction when data is partially received. This is done by calling abort() method on xmlHttp object when xmlHttp.readyState = 3. Ready state = 3 indicates that the Partial data has been received, but this content is not available yet.

We tested this behaviour on Firefox version 21.0 with objects of content size 500KB, 2MB, and 10MB on localhost. Objects with size 500KB and 2MB were cached in Firefox. But the object of size 10MB was not cached. As readyState is an event which notifies the state of the request, the abort() function gets called after the content has arrived in case of objects with size 500KB and 2MB. But when object with 10MB is used, abort() gets called before the content completely arrives. This leads to reception of broken resource and it is not cached by Firefox. We look at Firefox cached items by typing "about:cache" in the address bar to confirm this.

Chapter 6

Firefox version comparison based on caching implementation

6.1 Testing Environment

We downloaded Mozilla Firefox source code [6] for 19 versions starting from Firefox 3.6 till Firefox 21.0 which is the latest release as of writing this thesis. We compared the source files related to HTTP caching. We also referred to various bugzilla pages related to changes in caching. We identified the following HTTP caching related updates across these versions.

We have developed a Test bed for testing the important observations made through version code comparison and bugzilla. This includes a set of web-pages. Each web-page simulates a single scenario and we verify by comparing the results of execution of that web-page on related Firefox versions. We mention the details of the test case when we encounter the associated scenario.

6.2 Firefox 4

1. For checking if the content is partial, additional conditions are added in CheckCache function. These additional constraints which devoid the content from being partial are checking whether no-store directive is present in the response header; whether encoding is specified for the content; and whether the request is conditional.

2. SSL content was cached if the *disk_cache_ssl* preference was set or if the Cache-Control: public header was present. From version 4 onwards, it only checks for the preference.

6.3 Firefox 5

1. Earlier until Firefox 4, even if the content was loaded successfully, but cancel was called after the load succeeded, the cache entry was doomed. But, now the content is cached irrespective of whether the transaction was cancelled or not. It only depends on whether the content was loaded completely or not. [15] contains the bugzilla report for the change.

   In the test page, we use an XHR and call abort() to cancel the transaction after the response is completely received. In Firefox 4.0, this does not save the object in the cache. In Firefox 5.0 and ahead, this operation saves the content in cache.

6.4 Firefox 7

1. In determining whether the resource can be used from the cache, Vary headers are given higher priority than check_doc_frequency user preferences, validation check mechanisms based on no-cache & no-store, and response status codes.

6.5 Firefox 16

1. Does not return 304 responses from the cache, and also does not return any other non-redirect 3XX responses from the cache

6.6 Changes in user preferences

Following are few other preferences that have changed in caching:

- In Firefox 4, the accessibility.disablecache preference is removed; it was only exposed for debugging
- In Firefox 9, the default maximum size of an item in the disk cache was increased to 50 MB; previously, only items up to 5 MB were cached.

These preference changes are logged in Firefox release notes [16].

Chapter 7

Conclusion

We presented CacheMeter, a tool that calculates the percentage of objects encountered on a website that are marked browser-cacheable. CacheMeter uses directions given in RFC 2616 to determine the caching status of all the resources, by checking the HTTP Response Headers. We only consider browser caching parameters of the HTTP Response headers and do not look into intermediate proxy caching parameters. CacheMeter can be used by Web Administrators to determine the browser cacheability statistics for single or multiple webpages.

We ran CacheMeter on Alexa top 1000 websites. For calculating the results, we considered only those 864 websites which had atleast 50 HTTP Requests in our dataset. Our results indicate that on an average 77% of resources encountered across these 866 websites were cacheable. We identify resources like Images, Icons, JavaScript, JSON, CSS, and Application as type of objects which do not change frequently and hence cacheable. Our results show that 85% of such resources were explicitly marked as browser-cacheable. For HTTPS objects, the number of websites with atleast 50 HTTPS Requests was 196. 73% of those on an average were marked as browser-cacheable.

For images, we found that the cacheability percentage for size less than 100 bytes is below 53%. But cacheability percentage above 100 bytes increases to about 98%. A similar observation showing increase in cacheability percentage for larger objects can be made from Script and Text object results as well. Larger objects have more impact on network bandwidth and thus, if not cached, will affect user perceived latency. Results from CacheMeter meets this expectation.

We studied the flow of Firefox caching implementation to see how it implements RFC 2616 and also identified the changes encountered in various versions of Firefox with respect to caching. We found that Firefox caches every resource that is completely received. Firefox employs the RFC specified caching algorithm while making a choice of whether to use the entity from cache, when a subsequent request for the same resource arrives.

We look at some of the scenarios like when a broken resource is encountered or when the HTTP transaction is cancelled and see how Firefox handles these scenarios. Firefox implementation of caching makes sure that only complete responses are cached. If the content is incomplete, in case of broken resources, then the associated cache entry is doomed. So, if a transaction abruptly ends, but a HTTP resource is completely transmitted, then Firefox will cache the resource for future consideration. We verified these observations from Firefox implementation by writing webpages to test the scenarios.

The approach taken by Firefox maximizes the amount of objects that can be loaded from cache and also eliminates the risk of saving incomplete cache records due to some error in transaction. This approach gives good value to the cacheability employed by the websites.

Bibliography

[1] http://www.alexa.com

[2] http://www.internetworldstats.com/emarketing.htm

[3] http://www.ietf.org/rfc/rfc2616.txt

[4] Barish Greg, & Obraczka Katia (2000). *World Wide Web Caching: Trends and Techniques*. IEEE Communications Magazine

[5] Gwertzman James, & Seltzer Margo (1996). *World Wide Web Cache Consistency*. In Proceedings of the 1996 USENIX TECHNICAL CONFERENCE

[6] https://developer.mozilla.org/

[7] http://www.web-caching.com/personal-caches.html

[8] http://www-archive.mozilla.org/projects/netlib/http/http-caching-faq.html

[9] http://blog.httpwatch.com/2008/10/15/two-important-differences-between-firefox-and-ie-caching

[10] https://addons.mozilla.org/en-us/firefox/addon/http-request-logger

[11] https://code.google.com/p/selenium/wiki/PythonBindings

[12] http://code.activestate.com/recipes/576551-simple-web-crawler/

[13] http://mxr.mozilla.org/

[14] https://bugzilla.mozilla.org/show_bug.cgi?id=759043

[15] https://bugzilla.mozilla.org/show_bug.cgi?id=482935

[16] https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases

[17] https://developer.mozilla.org/en-US/docs/Mozilla/Debugging/HTTP_logging