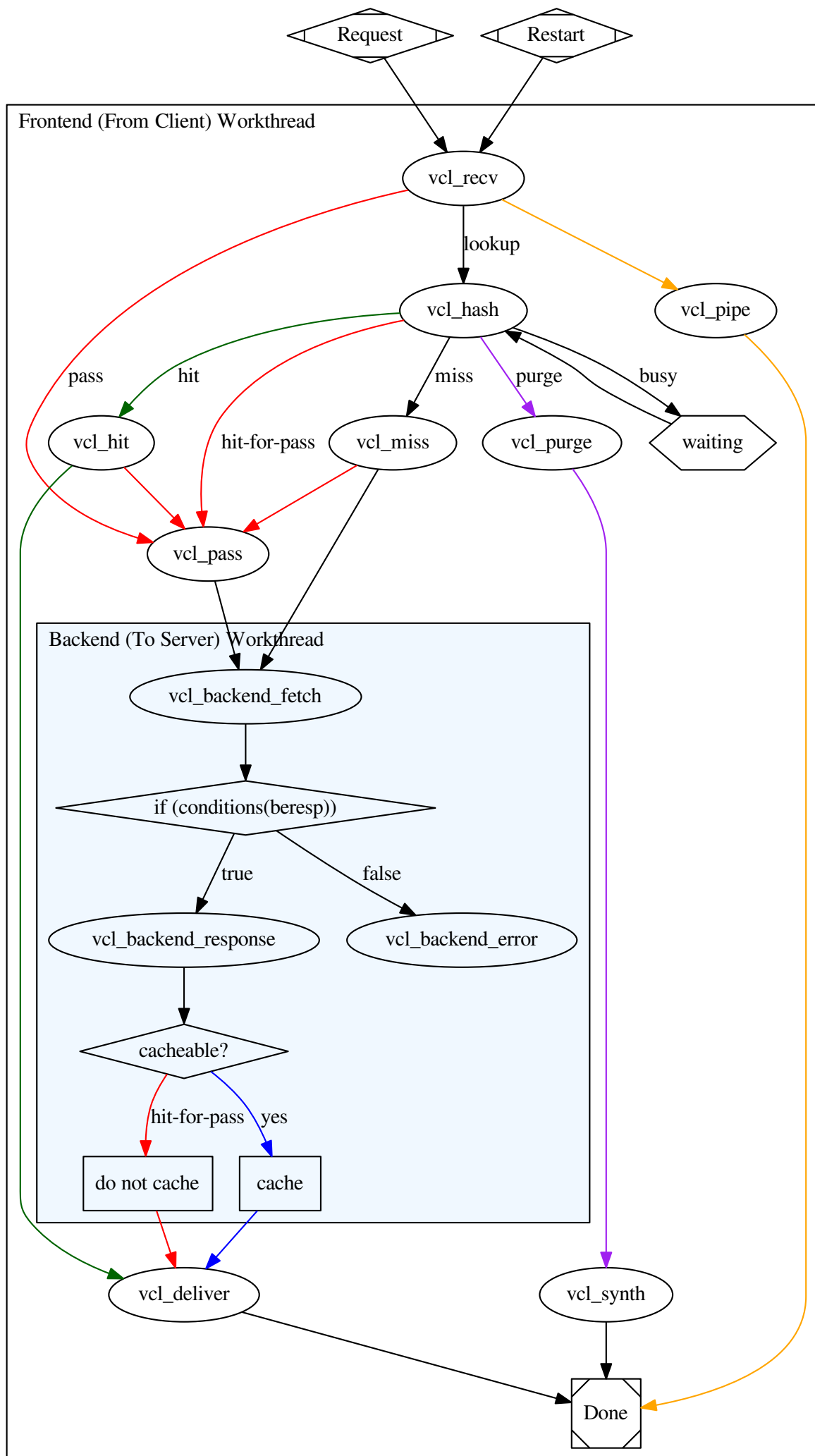
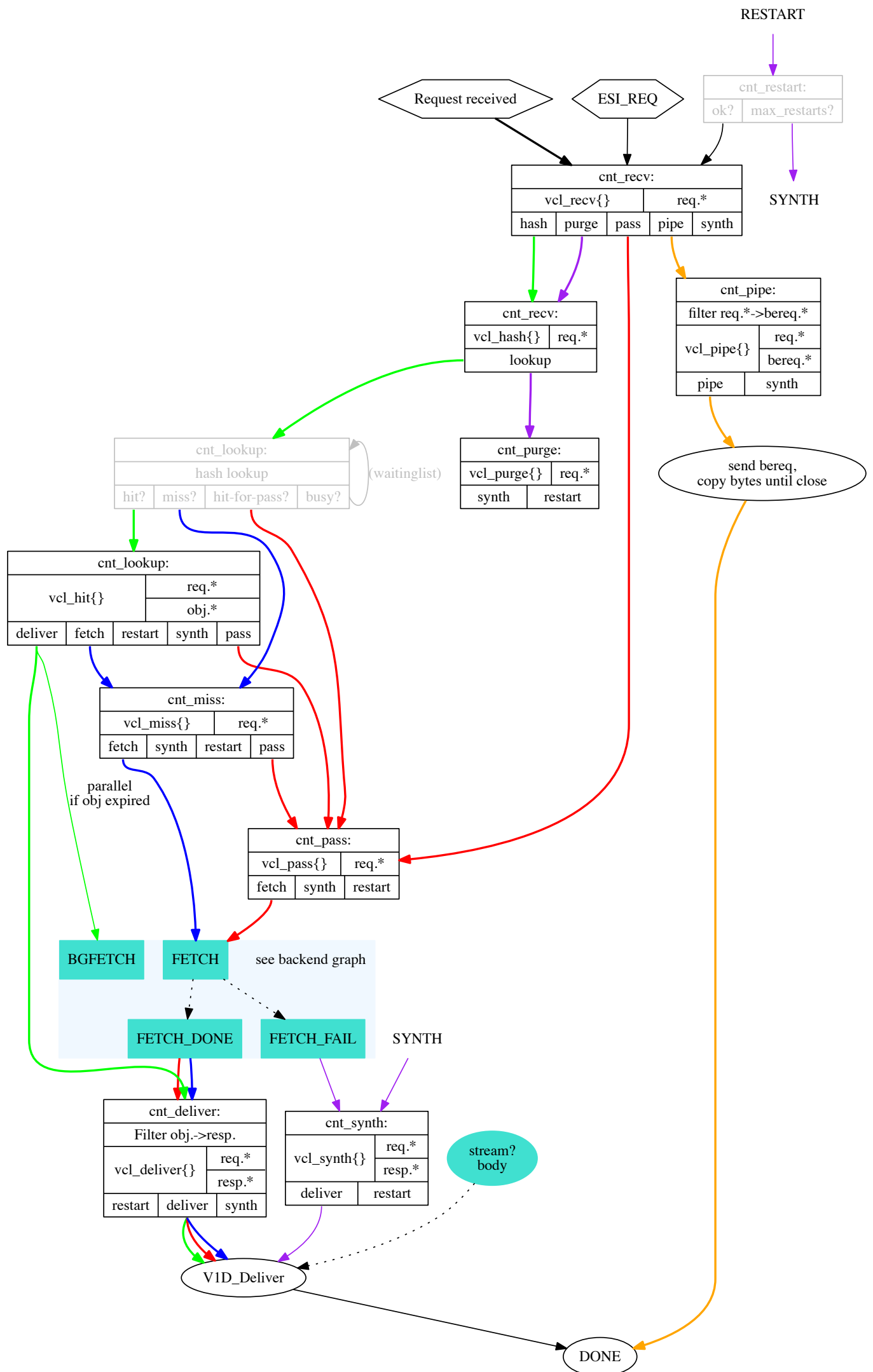
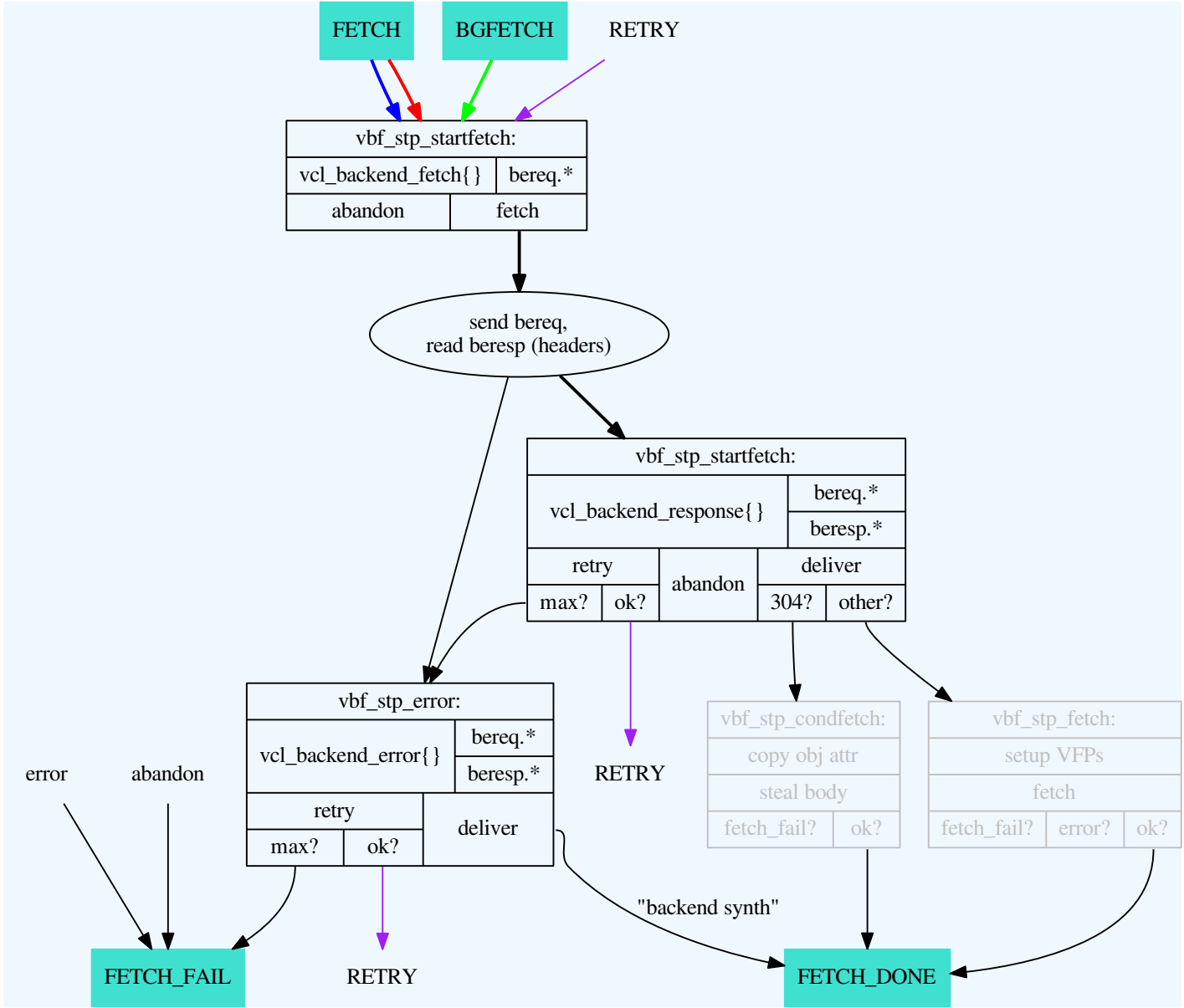




The Varnish Book







Authors: Francisco Velázquez ([Varnish Software](#)), Kristian Lyngstøl, Tollef Fog Heen, Jérôme Renard

Copyright: Varnish Software AS 2010-2015, Redpill Linpro AS 2008-2009

Versions: Documentation version-4.7-277-ge7c6276 / Tested for Varnish plus-4.0.3r1

Date: 2015-07-14

License: The material is available under a CC-BY-NC-SA license. See <http://creativecommons.org/licenses/by-nc-sa/3.0/> for the full license. For questions regarding what we mean by non-commercial, please contact training@varnish-software.com.

Contact: For any questions regarding this training material, please contact training@varnish-software.com.

Web: <http://www.varnish-software.com/book/>

Source: <http://github.com/varnish/Varnish-Book/>

Contents

1	Introduction	17
1.1	What is Varnish?	18
1.2	Varnish Cache and Varnish Plus	19
1.3	Varnish Software	21
1.4	What Is New in Varnish 4	22
2	Design Principles	24
2.1	How objects are stored	26
2.2	Object Lifetime	27
3	Getting Started	28
3.1	Varnish Distribution	29
3.2	Install Varnish and Apache as Backend	31
3.2.1	Install Apache	32
3.2.2	Install Varnish	33
3.2.3	Configure Varnish	35
3.2.4	Installation Test	37
3.3	The Management Interface <code>varnishadm</code>	38
3.4	More About Varnish Configuration	40
3.5	Command Line Configuration	42
3.6	Defining a Backend in VCL	44
3.7	Exercise: Use the administration interface to learn, review and set Varnish parameters	45
3.8	Exercise: Fetch data through Varnish	46
4	Examining Varnish Server's Output	47
4.1	Log Data Tools	48
4.2	Log Layout	49
4.3	Transactions	50
4.3.1	Transaction Groups	51
4.3.2	Example of Transaction Grouping with <code>varnishlog</code>	52
4.4	Query Language	53
4.5	Exercise	55
4.6	<code>varnishstat</code>	56
4.6.1	Notable Counters	60
4.7	Exercise: Try <code>varnishstat</code> and <code>varnishlog</code> together	62
5	Tuning	63
5.1	Varnish Architecture	64
5.1.1	The Parent Process: The Manager	66
5.1.2	The Child Process: The Cacher	67

5.1.3 VCL Compilation	68
5.2 Storage Backends	69
5.3 The SHared Memory Log (SHMLOG)	70
5.4 Tunable Parameters	71
5.5 Varnish Tuner	73
5.5.1 Varnish Tuner Persistence	74
5.5.2 Install Varnish Tuner	75
5.6 Threading Model	76
5.7 Threading parameters	77
5.7.1 Details of Threading Parameters	79
5.7.2 Number of Threads	80
5.7.3 Timing Thread Growth	81
5.8 System Parameters	82
5.9 Timers	83
5.10 Exercise: Tune first_byte_timeout	85
5.11 Exercise: Configure threading	86
6 HTTP	87
6.1 Protocol basics	88
6.2 Requests	89
6.3 Request example	90
6.4 Response	91
6.5 Response example	92
6.6 HTTP request/response control flow	93
6.7 Statelessness and idempotence	94
6.8 Cache related headers	95
6.9 Exercise: Test various Cache headers	96
6.10 Expires	97
6.11 Cache-Control	98
6.12 Last-Modified	100
6.13 If-Modified-Since	101
6.14 If-None-Match	102
6.15 Etag	103
6.16 Pragma	104
6.17 Vary	105
6.18 Age	106
6.19 Header availability summary	107
6.20 Cache-hit and misses	108
6.21 Exercise: Use <i>article.php</i> to test <i>Age</i>	109

7 VCL Basics	110
7.1 Varnish Finite State Machine	111
7.1.1 Waiting State	113
7.2 Detailed Varnish Request Flow for the Client Worker Thread	114
7.3 The VCL Finite State Machine	116
7.4 VCL Syntax	117
7.5 VCL built-in functions	118
7.6 Legal Return Actions	119
7.7 Variables in VCL subroutines	120
7.8 Summary of VCL	122
8 VCL Built-in Subroutines	123
8.1 VCL – <code>vcl_recv</code>	124
8.1.1 Built-in: <code>vcl_recv</code>	126
8.1.2 Example: Basic Device Detection	127
8.1.3 Exercise: Rewrite URLs and Host headers	128
8.1.4 Solution: Rewrite URLs and Host headers	130
8.2 VCL – <code>vcl_pass</code>	131
8.2.1 hit-for-pass	132
8.3 VCL – <code>vcl_backend_fetch</code> and <code>vcl_backend_response</code>	133
8.3.1 <code>vcl_backend_response</code>	135
8.3.2 The Initial Value of <code>beresp.ttl</code>	136
8.3.3 Example: Setting TTL of .jpg URLs to 60 seconds	138
8.3.4 Example: Cache .jpg for 60 seconds only if <code>s-maxage</code> is not present	139
8.3.5 Exercise: Avoid caching a page	140
8.3.6 Solution: Avoid caching a page	141
8.3.7 Exercise: Either use <code>s-maxage</code> or set TTL by file type	142
8.3.8 Solution: Either use <code>s-maxage</code> or set <code>ttl</code> by file type	143
8.4 VCL – <code>vcl_hash</code>	144
8.5 VCL – <code>vcl_hit</code>	145
8.6 VCL – <code>vcl_miss</code>	146
8.7 VCL – <code>vcl_deliver</code>	147
8.8 VCL – <code>vcl_synth</code>	148
8.8.1 Example: Redirecting requests with <code>vcl_synth</code>	150
8.9 Exercise: Modify the HTTP response header fields	151
8.9.1 Solution: Modify the HTTP response header fields	152
8.10 Exercise: Change the error message	153
8.10.1 Solution: Change the error message	154
9 Cache Invalidation	155

9.1	HTTP PURGE	157
9.1.1	VCL – <code>vcl_purge</code>	158
9.1.2	Example: PURGE	159
9.1.3	Exercise: PURGE an article from the backend	160
9.1.4	Solution: PURGE an article from the backend	161
9.2	PURGE with <code>restart</code> return action	164
9.3	Banning	166
9.3.1	Lurker-Friendly Bans	169
9.4	Exercise: Write a VCL program using <i>purge</i> and <i>ban</i>	171
9.4.1	Solution: Write a VCL program using <i>purge</i> and <i>ban</i>	172
9.5	Force Cache Misses	173
9.6	Hashtwo (Varnish Software Implementation of Surrogate Keys)	174
9.6.1	VCL example using Hashtwo	175
9.7	Purge vs. Bans vs. Hashtwo vs. Cache Misses	177
10	Saving a Request	179
10.1	Directors	180
10.1.1	Random Directors	182
10.2	Health Checks	183
10.3	Grace Mode	185
10.3.1	Time-line example	187
10.3.2	When can grace happen	188
10.3.3	Exercise: Grace	189
10.4	<code>retry</code> return action	190
10.5	Tune Backend Properties	191
10.6	Access Control Lists (ACLs)	192
11	Content Composition	193
11.1	A typical website	194
11.2	Cookies	195
11.2.1	Vary and Cookies	196
11.2.2	Best Practices for Cookies	197
11.2.3	Exercise: Compare Vary and <code>hash_data</code>	198
11.3	Edge Side Includes	199
11.3.1	Basic ESI usage	200
11.3.2	Example: Using ESI	201
11.3.3	Exercise: Enable ESI and Cookies	202
11.3.4	Testing ESI without Varnish	203
11.4	Masquerading AJAX requests	204
11.4.1	Exercise: write a VCL that masquerades XHR calls	205

11.4.2	Solution: write a VCL that masquerades XHR calls	206
12	Varnish Plus Software Components	207
12.1	Varnish Administration Console (VAC)	208
12.1.1	Overview Page of the Varnish Administration Console	209
12.1.2	Configuration Page of the Varnish Administration Console	210
12.1.3	Banning Page of the Varnish Administration Console	211
12.2	Varnish Custom Statistics (VCS)	212
12.2.1	VCS Data Model	214
12.2.2	VCS API	217
12.2.3	Screenshots of GUI	219
12.3	Varnish High Availability (VHA)	220
12.4	SSL/TLS Support	221
13	Appendix A: Resources	222
14	Appendix B: Varnish Programs	223
14.1	<code>varnishtop</code>	224
14.2	<code>varnishncsa</code>	225
14.3	<code>varnishhist</code>	226
14.4	Exercise: Try the tools	227
15	Appendix C: Extra Material	228
15.1	<code>ajax.html</code>	229
15.2	<code>article.php</code>	230
15.3	<code>cookies.php</code>	231
15.4	<code>esi-top.php</code>	232
15.5	<code>esi-user.php</code>	233
15.6	<code>httpheadersexample.php</code>	235
15.7	<code>purgearticle.php</code>	238
15.8	<code>test.php</code>	239
15.9	<code>set-cookie.php</code>	240
15.10	VCL Migrator from Varnish 3 to Varnish 4	241
16	Appendix D: Varnish Three Letter Acronyms	242

Abstract

The Varnish Book is the training material for Varnish Plus courses. The book teaches technical staff how to use Varnish Cache 4 and selected modules of Varnish Plus effectively.

The book explains how to get started with Varnish, and its Varnish Configuration Language (VCL). Covered are such procedures to effectively use VCL functions, cache invalidation, and more. Also included are Varnish utility programs such as `varnishlog`, and extra material.

Preface

- Course for Varnish Plus
- Learn specific features depending the course and your needs
- Necessary Background
- How to Use the Book
- Acknowledgements

After finishing this course, you will be able to install and configure the Varnish server, and write effective VCL code. The Varnish Book is designed for attendees of Varnish Plus courses. Varnish Plus is a commercial suite by Varnish Software that offers products for scalability, customization, monitoring, and expert support services. The engine of Varnish Plus is Varnish Cache Plus, which is the enhanced commercial edition of Varnish Cache.

Varnish Cache Plus should not be confused with Varnish Plus, a product offering by Varnish Software. Varnish Cache Plus is one of the software components available for Varnish Plus customers.

Most of the presented material in this book applies to both, the open source Varnish Cache and the commercial edition Varnish Cache Plus. Therefore, you can also refer to the Varnish Cache documentation at <https://www.varnish-cache.org/docs/4.0/>.

For simplicity, the book refers to *Varnish Cache* or *Varnish Cache Plus* as **Varnish** when there is no difference between them. There is more information about differences between Varnish Cache and Varnish Cache Plus in the [Varnish Cache and Varnish Plus](#) chapter.

The goal of this book is to make you confident when using Varnish. Varnish instructors focus on your area, needs or interest. Varnish courses are usually flexible enough to make room for it.

The instructor will cover selected material for the course you take. The System Administration (Admin) course provides attendees with the necessary knowledge to troubleshoot and tune common parameters of a Varnish server. The Web Developer (Webdev) course teaches how to adapt web applications so that they work with Varnish, which guarantees a fast experience for visitors of any website. Besides that, other courses may also be taught with this book.

Necessary Background

The Admin course requires that you:

- have expertise in a shell on a Linux/UNIX machine, including editing text files and starting daemons,
- understand HTTP cache headers,
- understand regular-expressions, and
- be able to install the software listed below.

The Webdev course requires that you:

- have expertise in a shell on a Linux/UNIX machine, including editing text files and starting daemons,
- understand HTTP cache headers,
- understand regular-expressions, and
- be able to install the software listed below.

You do not need background in theory or application behind Varnish to complete this course. However, it is assumed that you have experience and expertise in basic UNIX commands, and that you can install the following software:

- Varnish Cache 4.x or Varnish Cache Plus 4.x,
- Apache/2.4 or later,
- HTTPie 0.8.0 or later,
- PHP 5.4 or later, and
- curl – command line tool for transferring data with URL syntax-

More specific required skills depend on the course you take. The book starts with the installation of Varnish and navigation of some of the common configuration files. This part is perhaps the most UNIX-centric part of the course.

How to Use the Book

- Most of the material in this book applies to both: Varnish Cache and Varnish Cache Plus. Parts that apply only to Varnish Cache Plus are clearly stated.
- Varnish caching mechanisms are different than in other caching technologies. Open your mind and try to think different when using Varnish.
- The instructor guides you through the book.
- Use the *manual pages* and help options.
- See [Appendix D: Varnish Three Letter Acronyms](#) for a list of acronyms.

The Varnish Book is designed to be used as training material under the Varnish Plus course taught by a certified instructor. Under the course, the instructor guides you and selects the relevant sections to learn. However, you can also use this book as self-instructional material.

There are almost always many ways to do an exercise. The solutions provided in this book or by your instructor are not necessarily better than your own.

Varnish installs several reference manuals that are accessible through the manual page command `man`. You can issue the command `man -k varnish` to list the manual pages that mention Varnish in their short description. In addition, the `vs1` man page that explains the Varnish Shared Memory Logging (VSL). This man page does not come out when issuing `man -k varnish`, because it does not contain the word *varnish* in its short description.

The command `man varnishd`, for example, retrieves the manual page of the Varnish HTTP accelerator daemon. Also, some commands have a help option to print the usage of the command. For example, `varnishlog -h` prints the usage and options of the command with a short description of them.

In addition, you should refer to the documentation of Varnish Cache and Varnish Cache Plus. This documentation provides you extended details on the topics covered in this book and more. To access to this documentation, please visit <https://www.varnish-software.com/resources>.

The Varnish installation described in this book uses Ubuntu Linux 14.04 LTS (trusty), therefore most of the commands instructed in this book are for this Linux distribution. We point out some differences on how to configure Varnish for other Linux distributions, but you should reference your linux distribution's documentation for more details.

The book is written with different formatting conventions. Varnish Configuration Language (VCL) code uses the monospaced font type inside boxes:

```
vcl 4.0;

backend default {
    .host = "127.0.0.1";
    .port = "8080";
}

sub vcl_recv {
    # Do request header transformations here.
    if (req.url ~ "/admin") {
        return(pass);
    }
}
```

The first occurrence of a new term is usually its *definition*, and appears in the same font as the previous occurrence of "definition" in this sentence. File names are indicated like this:

/path/to/yourfile. Important notes, tips and warnings are also inside boxes, but they use the normal bodytext font type.

Resources, and Errata

- <https://varnish-cache.org>
- <https://varnish-software.com/academy>
- *#varnish-hacking* and *#varnish* on *irc.linpro.net*.
- <https://github.com/varnish/Varnish-Book/tree/Varnish-Book-v4>
- <https://www.varnish-cache.org/docs/trunk/users-guide/troubleshooting.html>

This book is meant to be understandable to everyone who takes a Varnish Plus course and has the required skills. If you find something unclear, do not be shy and blame yourself, ask your instructor for help. You can also contact the Varnish open source community at <https://varnish-cache.org>. To book training, please look at <https://varnish-software.com/academy>.

For those interested in development, the developers arrange weekly bug washes where recent tickets and development is discussed. This usually takes place on Mondays around 13:00 CET on the IRC channel *#varnish-hacking* on *irc.linpro.net*.

Errata, updates and general improvements of this book are available at its repository <https://github.com/varnish/Varnish-Book/tree/Varnish-Book-v4>.

Acknowledgements

In addition to the authors, the following deserve special thanks (in no particular order):

- Rubén Romero
- Dag Haavi Finstad
- Martin Blix Grydeland
- Reza Naghibi
- Federico G. Schwindt
- Dridi Boukelmoune
- Lasse Karstensen
- Per Buer
- Sevan Janiyan
- Kacper Wysocki
- Magnus Hagander
- Poul-Henning Kamp
- Everyone who has participated on the training courses

1 Introduction

- What is Varnish?
- Benefits of Varnish
- Open source / Free software
- Varnish Software: The company
- What is Varnish Plus?
- Varnish: more than a cache server
- History of Varnish
- Varnish Governance Board (VGB)



1.1 What is Varnish?

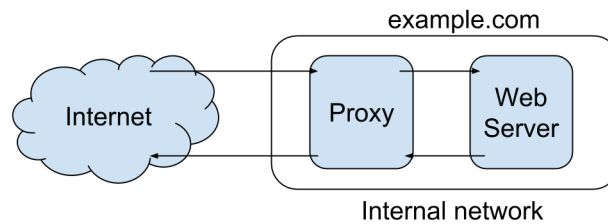


Figure 1: Reverse Proxy

Varnish is a reverse HTTP proxy, sometimes referred to as an HTTP accelerator or a web accelerator. A reverse proxy is a proxy server that appears to clients as an ordinary server. Varnish stores files or fragments of files in memory, allowing them to be served quickly. It is essentially a key/value store, that usually uses the URL as a key. It is designed for modern hardware, modern operating systems and modern work loads.

At the same time, Varnish is flexible. The Varnish Configuration Language (VCL) compiles and executes lightning fast. VCL allows system administrators to express their wanted policy rather than being constrained by what the Varnish developers want to cater for or could think of. Varnish has shown itself to work well both on large (and expensive) servers and tiny appliances.

Varnish is more than a reverse HTTP proxy. Depending on the installation, Varnish can be used as:

- web application firewall,
- DDoS attacks defender,
- load balancer,
- integration point,
- single sign-on gateway,
- authentication and authorization policy mechanism,
- quick fix for unstable backends, and
- HTTP router.



1.2 Varnish Cache and Varnish Plus

Table 1: Topics Covered in This Book and Their Availability in Varnish Cache and Varnish Plus

Topic	Varnish Cache	Varnish Plus
VCL	Yes	Yes
varnishlog	Yes	Yes
varnishadm	Yes	Yes
varnishncsa	Yes	Yes
varnishstat	Yes	Yes
varnishhist	Yes	Yes
varnishtest	Yes	Yes
varnishtop	Yes	Yes
directors	Yes	Yes
purge	Yes	Yes
ban	Yes	Yes
force cache misses	Yes	Yes
vagent2	Yes	Yes
Massive Storage Engine (MSE)	No	Yes
Varnish Administration Console (VAC)	No	Yes
Varnish Tuner	No	Yes
Hashtwo (Varnish Software Implementation of Surrogate Keys)	No	Yes
Varnish Custom Statistics (VCS)	No	Yes
Varnish High Availability (VHA)	No	Yes
SSL/TLS Support	No	Yes

Varnish Cache is an open source project, and free software. The development process is public and everyone can submit patches, or just take a peek at the code if there is some uncertainty on how does Varnish Cache work. There is a community of volunteers who help each other and newcomers. The BSD-like license used by Varnish Cache does not place significant restriction on re-use of the code, which makes it possible to integrate Varnish Cache in virtually any solution.

Varnish Cache is developed and tested on GNU/Linux and FreeBSD. The code-base is kept as self-contained as possible to avoid introducing out-side bugs and unneeded complexity. Therefore, Varnish uses very few external libraries.



Varnish Software is the company behind Varnish Cache. Varnish Software and the Varnish community maintain a package repository of Varnish Cache for several common GNU/Linux distributions.

Varnish Software also provides a commercial suite called Varnish Plus with software products for scalability, customization, monitoring and expert support services. The engine of the Varnish Plus commercial suite is the enhanced commercial edition of Varnish Cache. This edition is proprietary and it is called *Varnish Cache Plus*.

Table 1 shows the components covered in this book and their availability for Varnish Cache users and Varnish Plus customers. The covered components of Varnish Plus are described in the [Varnish Plus Software Components](#) chapter. For more information about the complete Varnish Plus offer, please visit <https://www.varnish-software.com/what-is-varnish-plus>.

At the moment of writing this book, Varnish Cache supports the operating systems and Linux distributions listed in Table 2.

Table 2: Varnish Cache and Varnish Plus supported platforms

	Varnish Cache	Varnish Plus
RedHat Enterprise Linux 5	Deprecated	Deprecated
RedHat Enterprise Linux 6	Yes	Yes
RedHat Enterprise Linux 7	Coming soon	Coming soon
Ubuntu Linux 12.04 LTS (precise)	Yes	Yes
Ubuntu Linux 14.04 LTS (trusty)	Yes	Yes
Debian Linux 7 (wheezy)	Yes	Yes
FreeBSD 9	Yes	No
FreeBSD 10	Yes	No

Varnish Cache and Varnish Plus support only 64-bit systems.

Note

Varnish Cache Plus should not be confused with Varnish Plus, a product offering by Varnish Software. Varnish Cache Plus is one of the software components available for Varnish Plus customers.



1.3 Varnish Software

Varnish timeline:

- 2005: Ideas! Verdens Gang (www.vg.no, Norway's biggest newspaper) were looking for alternative cache solutions.
- 2006: Work began. Redpill Linpro was in charge of project management, infrastructure and supporting development. Poul-Henning Kamp did the majority of the actual development.
- 2006: Varnish 1.0 released
- 2008: Varnish 2.0 released
- 2009: The first Varnish User Group Meeting is held in London. Roughly a dozen people participate from all around the world.
- 2010: Varnish Software is born as a spin-off to Redpill Linpro AS.
- 2011: Varnish 3.0 released
- 2012: The fifth Varnish User Group Meeting is held in Paris. Roughly 70 people participate on the User-day and around 30 on the developer-day!
- 2014: Varnish 4.0 released

VG, a large Norwegian newspaper, initiated the Varnish project in cooperation with Linpro. The lead developer of the Varnish project, Poul-Henning Kamp, is an experienced FreeBSD kernel hacker. Poul-Henning Kamp continues to bring his wisdom to Varnish in most areas where it counts.

From 2006 throughout 2008, most of the development was sponsored by VG, API, Escenic and Aftenposten, with project management, infrastructure and extra man-power provided by Redpill Linpro. At the time, Redpill Linpro had roughly 140 employees mostly centered around consulting services.

Today Varnish Software is able to fund the core development with income from service agreements, in addition to offering development of specific features on a case-by-case basis. The interest in Varnish continues to increase. An informal study based on the list of most popular web sites in Norway indicates that about 75% or more of the web traffic that originates in Norway is served through Varnish.

Varnish development is governed by the Varnish Governance Board (VGB), which thus far has not needed to intervene. The VGB consists of an architect, a community representative and a representative from Varnish Software.

As of December 2014, the VGB positions are filled by Poul-Henning Kamp (Architect), Rogier Mulhuijzen (Community) and Lasse Karstensen (Varnish Software). On a day-to-day basis, there is little need to interfere with the general flow of development.



1.4 What Is New in Varnish 4

- Version statement `vcl 4.0;`
- `req.request` is now `req.method`
- `vcl_fetch` is now `vcl_backend_response`
- Directors have been moved to the `vmod_directors`
- Hash directors as a client directors
- `vcl_error` is now `vcl_backend_error`
- `error()` is now `synth()`, and you must explicitly return it:
`return (synth(999, "Response"));`
- Synthetic responses in `vcl_synth`
- Setting headers on synthetic response bodies made in `vcl_synth` are now done on `resp.http` instead of `obj.http`.
- `obj.*` in `vcl_error` replaced by `beresp.*` in `vcl_backend_error`
- `hit_for_pass` objects are created using `beresp.uncacheable`
- `req.*` not available in `vcl_backend_response`
- `bereq.*` in `vcl_backend_response`
- `vcl_*` prefix reserved for builtin subroutines
- `req.backend.healthy` replaced by `std.healthy(req.backend_hint)`
- `client.port` and `server.port` replaced by `std.port(client.ip)` and `std.port(server.ip)`
- Cache invalidation with purges is now done via `return(purge)` in `vcl_recv`
- `obj.*` is now read-only
- `obj.last_use` is retired
- `vcl_recv` must now return `hash` instead of `lookup`
- `vcl_hash` must now return `lookup` instead of `hash`
- `vcl_pass` must now return `fetch` instead of `pass`
- `restart` in the backend is now `retry`, this is now called `return(retry)`, and jumps back up to `vcl_backend_fetch`
- *default* VCL is not called *builtin* VCL
- The builtin VCL now honors `Cache-Control: no-cache` (and friends) to indicate uncacheable content from the backend
- `remove` keyword replaced by `unset`
- `X-Forwarded-For` is now set before `vcl_recv`



- `session_linger` has been renamed to `timeout_linger` and it is in seconds now (previously was milliseconds)
- `sess_timeout` is renamed to `timeout_idle`
- Increasing `sess_workspace` is not longer necessary, you may need to increase either *workspace_backend* or *workspace_client*
- `thread_pool_purge_delay` is renamed to `thread_pool_destroy_delay` and it is in seconds now
- `thread_pool_add_delay` and `thread_pool_fail_delay` are in seconds now
- New parameter `vcc_allow_inline_c` to disable inline C in your VCL
- New query language to filter logs: `-m` option replaced by `-q`

The above list tries to summarize the most important changes from Varnish Cache 3 to Varnish Cache 4. For more information, please visit: <https://www.varnish-cache.org/docs/trunk/whats-new/upgrading.html>

If you want to migrate your VCL code from Varnish 3 to Varnish 4, you may be interested in looking at the *varnish3to4* script. See the [VCL Migrator from Varnish 3 to Varnish 4](#) section for more information.



2 Design Principles

Varnish is designed to:

- Solve real problems
- Run on modern hardware (64-bit multi-core architectures)
- Work with the kernel, not against it
- Translate Varnish Configuration Language (VCL) to C programming language
- Be extendible via Varnish Modules (VMODs)
- Reduce lock-contention via its workspace-oriented shared memory model

The focus of Varnish has always been performance and flexibility. Varnish is designed for hardware that you buy today, not the hardware you bought 15 years ago. This is a trade-off to gain a simpler design and focus resources on modern hardware. Varnish is designed to run on 64-bit architectures and scales almost proportional to the number of CPU cores you have available. Though CPU-power is rarely a problem.

32-bit systems, in comparison to 64-bit systems, allow you to allocate less amount of virtual memory space and less number of threads. The theoretical maximum space depends on the operating system (OS) kernel, but 32-bit systems usually are bounded to 4GB. You may get, however, about 3GB because the OS reserves some space for the kernel.

Varnish uses a workspace-oriented memory-model instead of allocating the exact amount of space it needs at run-time. Varnish does not manage its allocated memory, but it delegates this task to the OS because the kernel can normally do this task better than a user-space program.

Event filters and notifications facilities such as `epoll` and `kqueue` are advanced features of the OS that are designed for high-performance services like Varnish. By using these, Varnish can move a lot of the complexity into the OS kernel which is also better positioned to decide which threads are ready to execute and when.

In addition, Varnish uses a configuration language (VCL) that is translated to C programming language code. This code is compiled with a standard C compiler and then dynamically linked directly into Varnish at run-time. This has several advantages. The most practical is the freedom you get as system administrator.

You can use VCL to decide how you want to interact with Varnish, instead of having a developer trying to predict every possible caching scenario. The fact that VCL is translated to C code, gives Varnish a very high performance. You can also by-pass the process of code translation and write raw C code, this is called in-line C in VCL. In short: VCL allows you to specify exactly how to use and combine the features of Varnish.

Varnish allows integration of Varnish Modules or simply VMODs. These modules let you extend the functionality of VCL by pulling in custom-written features. Some examples include non-standard header manipulation, access to *memcached* or complex normalization of headers.

The shared memory log (SHMLOG) allows Varnish to log large amounts of information at almost no cost by having other applications parse the data and extract the useful bits. This design and other mechanisms decrease lock-contention in the heavily threaded environment of Varnish.



To summarize: Varnish is designed to run on modern hardware under real work-loads and to solve real problems. Varnish does not cater to the "I want to make Varnish run on my 486 just because"-crowd. If it does work on your 486, then that's fine, but that's not where you will see our focus. Nor will you see us sacrifice performance or simplicity for the sake of niche use-cases that can easily be solved by other means -- like using a 64-bit OS.



2.1 How objects are stored

- Objects in Varnish are stored in memory and addressed by hash keys
- You can control the hashing
- Multiple objects can have the same hash key

Varnish has a key/value store in its core. Objects are stored in memory and references to these objects are kept in a hash tree.

A rather unique feature of Varnish is that it allows you to control the input of the hashing algorithm. The key is by default made out of the HTTP Host header and the URL, which is sufficient and recommended for typical cases. However, you are able to create the key from something else. For example, you can use cookies or the user-agent of a client request to create a hash key.

HTTP specifies that multiple objects can be served from the same URL, depending on the preferences of the client. For instance, content in *gzip* format is sent only to clients that indicate *gzip* support. Varnish stores various objects under one key. Upon a client request, Varnish selects the object that matches the client preferences.



2.2 Object Lifetime

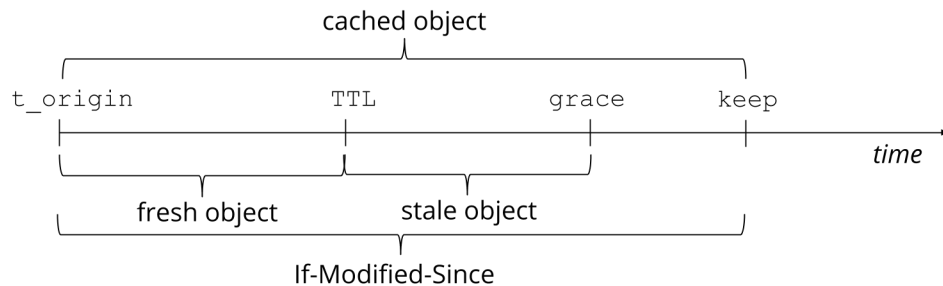


Figure 2: Object Lifetime

Figure 2 shows the lifetime of cached objects. A cached object has an origin timestamp t_{origin} and three duration attributes: 1) TTL, 2) grace, and 3) keep. t_{origin} is the time when an object was created in the backend. An object lives in cache until $\text{TTL} + \text{grace} + \text{keep}$ elapses. After that time, the object is removed by the Varnish daemon.

In a timeline, objects within the time-to-live TTL are considered *fresh objects*. *Stale objects* are those within the time period TTL and grace. Objects within t_{origin} and keep are used when applying conditions with the HTTP header If-Modified-Since.

The [VCL - vcl_backend_fetch](#) and [vcl_backend_response](#) section explains how Varnish handles backend responses and how these duration attributes affect subsequent actions.



3 Getting Started

In this chapter, you will:

- learn about the Varnish distribution,
- install Varnish and Apache,
- configure Varnish to use Apache as backend, and
- cover basic configuration.

Most of the commands you will type in this course require root privileges. You can get temporary root privileges by typing `sudo <command>`, or permanent root privileges by typing `sudo -i`.

In Varnish terminology, a backend server is whatever server Varnish talks to fetch content. This can be any sort of service as long as it understands HTTP. Most of the time, Varnish talks to a web server or an application frontend server. In this book, we use backend, web server or application frontend server interchangeably.



3.1 Varnish Distribution

Utility programs part of the Varnish distribution:

- `varnishd`
- `varnishadm`
- `varnishlog`
- `varnishstat`
- and more

The Varnish distribution includes several utility programs that you will use in this course. You will learn how to use these programs as you progress, but it is useful to have a brief introduction about them before we start.

The central block of Varnish is the Varnish daemon `varnishd`. This daemon accepts HTTP requests from clients, sends requests to a backend and caches the returned objects. `varnishd` is further explained in the [Varnish Architecture](#) section.

`varnishadm` controls a running Varnish instance. The `varnishadm` utility establishes a command line interface (CLI) connection to `varnishd`. This utility is the only one that may affect a running instance of Varnish. You can use `varnishadm` to:

- start and stop `varnishd`,
- change configuration parameters,
- reload the Varnish Configuration Language (VCL),
- view the most up-to-date documentation for parameters, and
- more.

The [Management Interface `varnishadm`](#) section explains in more detail this utility.

The Varnish log provides large amounts of information, thus it is usually necessary to filter it. For example, "show me only what matches X". `varnishlog` does precisely that. You will learn more about `varnishlog` in the [Examining Varnish Server's Output](#) chapter.

`varnishstat` is used to access **global counters**. It provides overall statistics, e.g the number of total requests, number of objects, and more. `varnishstat` is particularly useful when using it together with `varnishlog` to analyze your Varnish installation. The [varnishstat](#) section explains in detail this utility.

In addition, there are other utility programs such as `varnishncsa`, `varnishtop` and `varnishhist`. [Appendix B: Varnish Programs](#) explains them.

Note

There is a delay in the log process, but usually not noticeable.





3.2 Install Varnish and Apache as Backend

Use packages provided by

- varnish-software.com for Varnish Cache Plus
- varnish-cache.org for Varnish Cache

Table 3: Different Locations of the Varnish Configuration File

SysV	SysV	systemd	systemd
Ubuntu/Debian	RHEL/CentOS	Ubuntu/Debian	Fedora/RHEL 7+/CentOS 7+
/etc/default/varnish	/etc/sysconfig/varnish	/etc/systemd/system/varnish.service [1]	/etc/varnish/varnish.params
/etc/default/varnishlog	[2]	/etc/systemd/system/varnishlog.service [1]	[3]
/etc/default/varnishncsa	[2]	/etc/systemd/system/varnishncsa.service [1]	[3]

[1] The file does not exist by default. Copy it from `/lib/systemd/system/` and edit it.

[2] There is no configuration file. Use the command `chkconfig varnishlog/varnishncsa on/off` instead.

[3] There is no configuration file. Use the command `systemctl start/stop/enable/disable/ varnishlog/varnishncsa` instead.

Table 4: Varnish and Apache Configuration

Server	Result	Configuration file
Varnish	Listens on port 80	/etc/default/varnish *
Varnish	Uses Apache as backend on localhost:8080	/etc/varnish/default.vcl
Apache	Listens on port 8080	/etc/apache2/ports.conf * and /etc/apache2/sites-enabled/000-default *

* These files are for a SysV Ubuntu/Debian configuration.

The configuration file is used to give parameters and command line arguments to the Varnish daemon. This file also specifies the location of the VCL file. Modifications to this file require to run `service varnish restart` for the changes to take effect.

The location of the Varnish configuration file depends on the operating system and whether it uses the `init` system of *SysV*, or *systemd*. Table 3 shows the locations for each system installation.

To install packages on Ubuntu and Debian, use the command `apt-get install <package>`, e.g., `apt-get install varnish`. For Red Hat, use `yum install <package>`.



3.2.1 Install Apache

To install Apache in Ubuntu, type the command: `apt-get install apache2`. Install the *HTTPIe* utility with the command: `apt-get install httpie`. HTTPie allows you to issue arbitrary HTTP requests in the terminal. Next:

1. Verify that Apache works by typing `http -h localhost`. You should see a 200 OK response from Apache.
2. Change Apache's port from 80 to 8080 in `/etc/apache2/ports.conf` and `/etc/apache2/sites-enabled/000-default.conf`.
3. Restart Apache: `service apache2 restart`.
4. Verify that Apache still works by typing `http -h localhost:8080`.



3.2.2 Install Varnish

To use the **varnish-software.com** repository and install **Varnish Cache Plus 4** on Ubuntu 14.04 trusty do the following as root:

```
#. apt-get install apt-transport-https
#. apt-get install curl
#. curl https://<username>:<password>@repo.varnish-software.com/GPG-key.txt \
| apt-key add -
```

Add the repositories for Varnish Cache Plus and VMODs in `/etc/apt/sources.list.d/varnish-4.0-plus.list`:

```
# Remember to replace DISTRO and RELEASE with what applies to your system.
# distro=(debian|ubuntu), RELEASE=(precise|trusty|wheezy|jessie)

# Varnish Cache Plus 4.0 and VMODs
deb https://<username>:<password>@repo.varnish-software.com/DISTRO RELEASE \
varnish-4.0-plus

# non-free contains VAC, VCS, Varnish Tuner and proprietary VMODs.
deb https://<username>:<password>@repo.varnish-software.com/DISTRO RELEASE \
non-free
```

Then:

```
apt-get update
apt-get install varnish-plus
```

All software related to **Varnish Cache Plus** including VMODs are available in RedHat and Debian package repositories. These repositories are available on <http://repo.varnish-software.com/>, using your customer specific username and password.

Varnish is already distributed in many package repositories, but those packages might contain an outdated Varnish version. Therefore, we recommend you to use the packages provided by [varnish-software.com](http://repo.varnish-software.com/) for *Varnish Cache Plus* or varnish-cache.org for *Varnish Cache*. Please be advised that we only provide packages for LTS releases, not all the intermediate releases. However, these packages might still work fine on newer releases.

To use Varnish Cache Plus repositories on RHEL 6, put the following in `/etc/yum.repos.d/varnish-4.0-plus.repo`:

```
[varnish-4.0-plus]
name=Varnish Cache Plus
baseurl=https://<username>:<password>@repo.varnish-software.com/redhat
/varnish-4.0-plus/el$releasever
enabled=1
gpgcheck=0
```



```
[varnish-admin-console]
name=Varnish Administration Console
baseurl=
https://<username>:<password>@repo.varnish-software.com/redhat
/vac/el$releasever
enabled=1
gpgcheck=0

[varnishtuner]
name=Varnish Tuner
baseurl=
https://<username>:<password>@repo.varnish-software.com/redhat
/varnishtuner/el$releasever
enabled=1
gpgcheck=0
```

If you want to install **Varnish Cache** in Ubuntu change the corresponding above lines to:

```
curl https://repo.varnish-cache.org/ubuntu/GPG-key.txt | apt-key add -
echo "deb https://repo.varnish-cache.org/ubuntu/ trusty varnish-4.0" >> \
/etc/apt/sources.list.d/varnish-cache.list
apt-get install varnish
```

Change the Linux distribution and Varnish Cache release in the needed lines.



3.2.3 Configure Varnish

Configure the Varnish `DAEMON_OPTS`:

```
-a ${VARNISH_LISTEN_ADDRESS}:${VARNISH_LISTEN_PORT}
-T ${VARNISH_ADMIN_LISTEN_ADDRESS}:${VARNISH_ADMIN_LISTEN_PORT}
```

See [Table 3](#) and locate the Varnish configuration file for your installation. Open and edit that file to listen on port 80 and have a management interface on port 1234. This is configured with the variable `DAEMON_OPTS`, and options `-a` and `-T` respectively.

In order for changes in the configuration file to take effect, *varnishd* must be restarted. The safest way to restart Varnish is by using `service varnish restart`.

The default VCL file location is `/etc/varnish/default.vcl`. You can change this location by editing the configuration file. The VCL file contains your VCL and backend definitions. Edit the VCL file to use Apache as backend:

```
backend default {
    .host = "127.0.0.1";
    .port = "8080";
}
```

After changing a VCL file, you can run `service varnish reload`. This command does **not** restart *varnishd*, it only reloads the VCL code. The result of your configuration is resumed in [Table 4](#).

You can get an overview over services listening on TCP ports by issuing the command `netstat -nlpt`. Within the result, you should see something like:

tcp	0	0	0.0.0.0:80	0.0.0.0:*	LISTEN	9223/varnishd
tcp	0	0	127.0.0.1:1234	0.0.0.0:*	LISTEN	9221/varnishd

Note

We recommend you to disable Security-Enhanced Linux (SELinux). If you prefer otherwise, then set the boolean `varnishd_connect_any` variable to 1. You can do that by executing the command `sudo setsebool varnishd_connect_any 1`. Also, be aware that SELinux defines the ports 6081 and 6082 for *varnishd*.

Tip

Issue the command `man vcl` to see all available options to define a backend.



Tip

You can also configure Varnish via the [Varnish Administration Console \(VAC\)](#).

The screenshot displays the Varnish Administration Console (VAC) interface. The top navigation bar includes the VAC logo, tabs for OVERVIEW, CONFIGURE, BANS, USERS, SUPPORT, and MESSAGES, and a user status section showing 'Signed as vac' and a 'Sign out' button. The left sidebar has tabs for CACHES, VCL (selected), and PARAMETERS. Under the VCL tab, there is a list of VCLs with 'my_vcl' selected, a plus icon to add a new VCL, and a 'CREATE NEW VCL' button. The main content area features a 'CREATE NEW CACHE GROUP' button at the top. Below it is a 'Test' section with a pencil icon and an ID '553661f00cf2f2135d9f2c61'. This section contains two graphs: 'Instant request' (0.00 req/sec) and 'Hitrate' (0.00 hit/sec). Below the graphs are three sections: 'Group caches' with a list containing '127.0.0.1:6085' and a '4.0' value, with 'SAVE CHANGES' and 'UNDO CHANGES' buttons; 'VCL' with a list containing 'my_vcl' and a 'RESTART GROUP' button; and 'Parameter set' with a list containing 'my_parameter_set' and a 'DELETE GROUP' button.

Figure 3: GUI to configure Varnish via the [Varnish Administration Console \(VAC\)](#).



3.2.4 Installation Test

```
# http -p Hh localhost
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate, compress
Host: localhost
User-Agent: HTTPie/0.8.0

HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 0
Connection: keep-alive
Content-Encoding: gzip
Content-Length: 3256
Content-Type: text/html
Date: Wed, 18 Mar 2015 13:55:28 GMT
ETag: "2cf6-5118f93ad6885-gzip"
Last-Modified: Wed, 18 Mar 2015 12:53:59 GMT
Server: Apache/2.4.7 (Ubuntu)
Vary: Accept-Encoding
Via: 1.1 varnish-plus-v4
X-Varnish: 32770
```

You can test your Varnish installation by issuing the command `http -p Hh localhost`. If you see the HTTP response header field `via: 1.1 varnish-plus-v4`, then your installation is correct.



3.3 The Management Interface `varnishadm`

Varnish offers a management command line interface (CLI) to control a running Varnish instance. This interface implements a list of management commands in the `varnishadm` utility program. `varnishadm` establishes a connection to the Varnish daemon `varnishd`. You can use `varnishadm` to:

- start and stop the cacher (aka child) process
- change configuration parameters without restarting Varnish
- reload the Varnish Configuration Language (VCL) without restarting Varnish
- view the most up-to-date documentation for parameters

You can read about other usages by issuing the `help` command after you connect to the management interface (`varnishadm`).

To connect to the management interface, issue the command `varnishadm`. If there are many Varnish instances running in one machine, specify the instance with the `-n` option. Keep the following in mind when using the management interface:

1. Changes take effect on the running Varnish daemon instance without need to restart it.
2. Changes are not persistent across restarts of Varnish. If you change a parameter and you want the change to persist after you restart Varnish, you need to store your changes in the configuration file of the boot script. The location of the configuration file is in [Table 3](#)

`varnishadm` uses a non-encrypted key stored in a secret file to authenticate and connect to a Varnish daemon. You can now provide access to the interface on a per user basis by adjusting the read permission on the secret file. The location of the secret file is `/etc/varnish/secret` by default, but you can use the `-s` option to specify other location. The content of the file is a shared secret, which is a string generated under Varnish installation.

The management interface authenticates with a challenge-response mechanism. Therefore, the shared secret is never transmitted, but a challenge and the response to the challenge. This authentication mechanism offers a reasonably good access control, but it does not protect the data transmitted over the connection. Therefore, it is very important to avoid eavesdroppers like in the man-in-the-middle attack. The simplest way to avoid eavesdroppers is to configure the management interface listening address of `varnishd` to listen only on localhost (127.0.0.1). You configure this address with the `-T` option of the `varnishd` command.

Tip

Varnish provides many on-line reference manuals. To learn more about `varnishadm`, issue `man varnishadm`. To check the Varnish CLI manual page, issue `man varnish-cli`.



Tip

You can also access the `varnishadm` via the [Varnish Administration Console \(VAC\)](#). To do that, you just have to navigate to the *CONFIGURE* tab and click on the Varnish server you want to administrate. Then, `varnishadm` is ready to use in a terminal emulator right in your web browser.

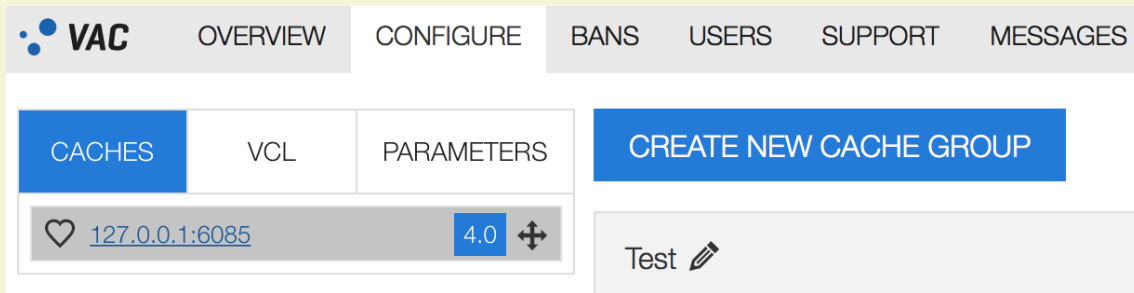


Figure 4: Access to `varnishadm` by clicking on the Varnish server that you want to administrate.

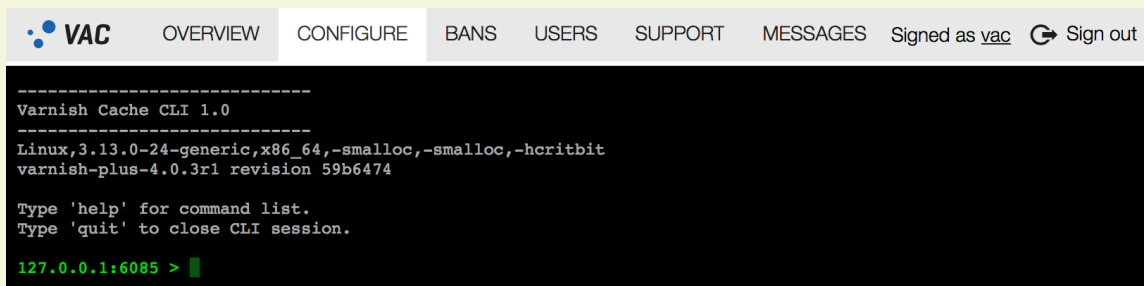


Figure 5: Terminal emulator in your web browser.



3.4 More About Varnish Configuration

Table 5: Varnish Configuration Types

Configuration Type	Restart Required	Persistence at next restart
Command line options	Yes	If stored in the configuration file as part of DAEMON_OPTS
Tunable parameters	No (if changed in varnishadm)	If stored in the configuration file as part of DAEMON_OPTS
Configuration in VCL	No	Yes

The location of the configuration file is in [Table 3](#).

Table 6: How to reload Varnish

Command	Result
<code>service varnish restart</code>	Restarts Varnish using the operating system mechanisms. Caches are flushed.
<code>service varnish reload</code>	Only reloads the VCL. Caches are not affected.
<code>varnishadm vcl.load <configname> <filename></code> and <code>varnishadm vcl.use <configname></code>	Can be used to manually reload VCL. The <code>service varnish reload</code> command does this for you automatically.
<code>varnishadm param.set <param> <value></code>	Can be used to set parameters without restarting Varnish.

Using the `service` commands is recommended, safe and fast.

Command line options and tunable parameters are used to: 1) define how Varnish should work with operating system and hardware, and 2) set default values. Configuration in VCL defines how Varnish should interact with web servers and clients.

Almost every aspect of Varnish can be reconfigured without restarting Varnish. Notable exceptions are: 1) allocated memory size for caching, 2) cache file location, 3) ownership (for user and group privileges) of the Varnish daemon, and 4) the hashing algorithm.

Some parameters changes require to restart Varnish to take effect. For example, the modification of the listening port. Other changes might not take effect immediately, but restart is not required. Changes to cache time-to-live (TTL), for instance, take effect only after the current cached objects expire. In this example, the value of the TTL parameter is only applicable to caches fetched after the TTL modification.

`param.show <parameter>` outputs a description of parameter. The description includes when and how modifications takes effect, and the default and current value of the parameter.



There are other ways to reload VCL and make parameter-changes take effect, mostly using the `varnishadm` tool. However, using the `service varnish reload` and `service varnish restart` commands is a good habit.

Note

If you want to know how the `service varnish-commands` work, look at the script that runs behind the scenes. The script is in `/etc/init.d/varnish`.

Warning

The `varnish` script-configuration (located under `/etc/default/` or `/etc/sysconfig/`) is directly sourced as a shell script. Pay close attention to any backslashes (`\`) and quotation marks that might move around as you edit the `DAEMON_OPTS` environmental variable.



3.5 Command Line Configuration

Relevant options for the course are:

<code>-a <[hostname]:port></code>	listening address and port for client requests
<code>-f <filename></code>	VCL file
<code>-p <parameter=value></code>	set tunable parameters
<code>-S <secretfile></code>	shared secret file for authorizing access to the management interface
<code>-T <hostname:port></code>	listening address and port for the management interface
<code>-s <storagetype,options></code>	where and how to store objects

All the options that you can pass to the `varnishd` binary are documented in the `varnishd(1)` manual page (`man varnishd`). You may want to take a moment to skim over the options mentioned above.

For Varnish to start, you must specify a backend. You can specify a backend by two means: 1) declare it in a VCL file, or 2) use the `-b` to declare a backend when starting `varnishd`.

Though they are not strictly required, you almost always want to specify a `-s` to select a storage backend, `-a` to make sure Varnish listens for clients on the port you expect and `-T` to enable a management interface, often referred to as a telnet interface.

For both `-T` and `-a`, you do not need to specify an IP, but can use `:80` to tell Varnish to listen to port 80 on all IPs available. Make sure you do not forget the colon, as `-a 80` tells Varnish to listen to the IP with the decimal-representation "80", which is almost certainly not what you want. This is a result of the underlying function that accepts this kind of syntax.

You can specify `-p` for parameters multiple times. The workflow for tuning Varnish parameters usually is that you first try the parameter on a running Varnish through the management interface to find the value you want. Then, you store the parameter and value in a configuration file. This file is read every time you start Varnish.

The `-S` option specifies a file which contains a secret to be used for authentication. This can be used to authenticate with `varnishadm -S` as long as `varnishadm` can read the same secret file -- or rather the same content: The content of the file can be copied to another machine to allow `varnishadm` to access the management interface remotely.

Note

Varnish requires that you specify a backend. A backend is normally specified in the VCL file. You specify the VCL file with the `-f` option. However, it is possible to start Varnish without a VCL file by specifying the backend server with the `-b <hostname:port>` option instead.

Since the `-b` option is mutually exclusive with the `-f` option, we use only the `-f` option. You can use `-b` if you do not intend to specify any VCL and only have a single backend server.



Tip

Type `man varnishd` to see all options of the Varnish daemon.



3.6 Defining a Backend in VCL

/etc/varnish/default.vcl

```
vcl 4.0;

backend default {
    .host = "localhost";
    .port = "8080";
}
```

The above example defines a backend named `default`, where the name *default* is not special. Varnish uses the first backend you specify as default. You can specify many backends at the same time, but for now, we will only specify one to get started.

Tip

You can also add and edit your VCL code via the [Varnish Administration Console \(VAC\)](#). This interface also allows you to administrate your VCL files.

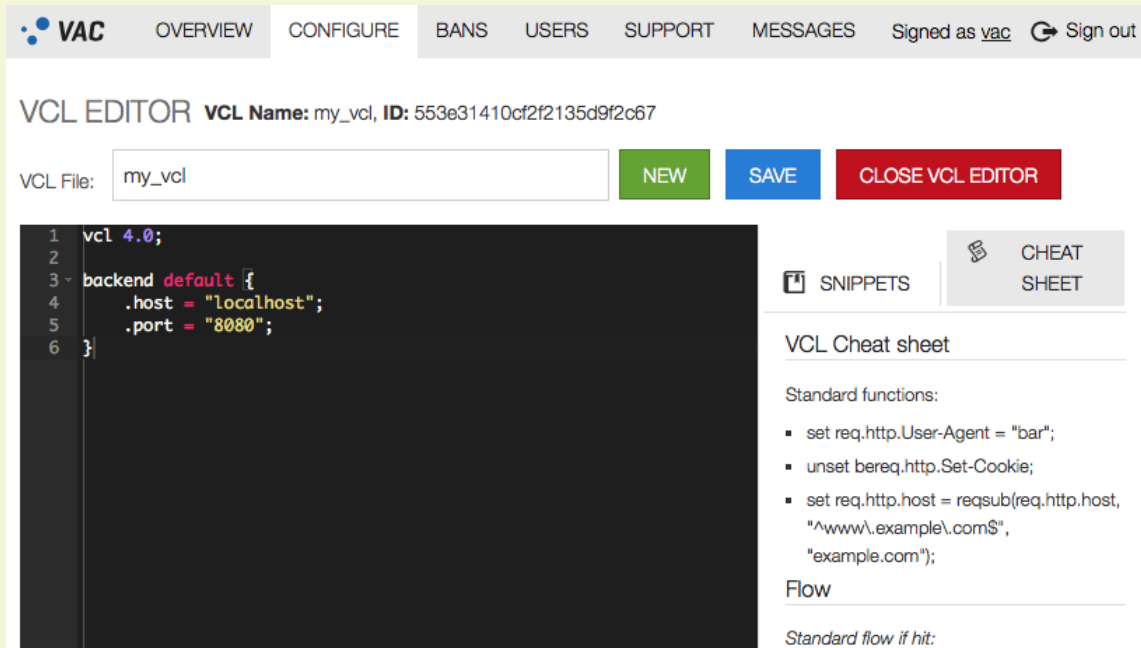


Figure 6: GUI of [Varnish Administration Console \(VAC\)](#) with command line interface to edit your VCL code.



3.7 Exercise: Use the administration interface to learn, review and set Varnish parameters

1. Use `varnishadm` to see the default value for the `default_ttl` parameter and what it does.



3.8 Exercise: Fetch data through Varnish

- Execute `http -p hH http://localhost/` on the command line
- Compare the results from multiple executions.

`-p hH` specifies HTTPie to print only request and response headers, but not the content. The typical HTTP response is "200 OK" or "404 File not found". Feel free to try removing some of the options observe the effect. For more information about the HTTPie command, type `man http`.

Testing Varnish with a web browser can be confusing, because web browsers have their own cache. Therefore, it is useful to double-check web browsers requests with HTTPie.



4 Examining Varnish Server's Output

In this chapter you will learn about:

- log records,
- statistics out from global counters and the Varnish log,
- the log layout,
- transactions,
- the query language, and
- notable counters.

Varnish provides log data in real-time, which is accessible through Varnish tools. Varnish logs all its information to [The SHared Memory LOG \(SHMLOG\)](#). This memory log is overwritten when filled-up in circular order.

The memory log overwriting has two effects. On the one hand, there is no historic data, but on the other hand, there is an abundance of information accessible at a very high speed. Still, you can instruct Varnish to store logs in files.

The `varnishlog` and `varnishncsa` configuration files allow you to enable or disable log writing to disk. Nevertheless, keep in mind that `varnishlog` generates large amounts of data! [Table 3](#) in the *Install Varnish and Apache as backend* section shows the location of the configuration file based on your platform.

Varnish provides specific tools to parse the content of logs: `varnishlog`, `varnishncsa`, and `varnishstat` among others. `varnishlog` and `varnishstat` are the two most common used tools.

Tip

All utility programs to display Varnish logs have installed reference manuals. Use the `man` command to retrieve their manual pages.



4.1 Log Data Tools

Tools to display detailed log records:

- `varnishlog` is used to access request-specific data. It provides information about specific clients and requests.
- `varnishncsa` displays Varnish access logs in NCSA Common log format.

Statistical tools:

- `varnishstat` is used to access **global counters**.
- `varnishtop` reads the Varnish log and presents a continuously updated list of the most commonly occurring log entries.
- `varnishhist` reads the Varnish log and presents a continuously updated histogram showing the distribution of the last *N* requests by their processing.

If you have multiple Varnish instances on the same machine, you need to specify `-n <name>` both when starting Varnish and when using the tools. This option is used to specify the instance of `varnishd`, or the location of the shared memory log. All tools (including `varnishadm`) can also take a `-n` option.

In this course, we focus on the two most important tools: `varnishlog` and `varnishstat`. Unlike all other tools, `varnishstat` does not read entries from the Varnish log, but from global counters. You can find more details about the other Varnish tools `varnishncsa`, `varnishtop` and `varnishhist` in [Appendix B: Varnish Programs](#).



4.2 Log Layout

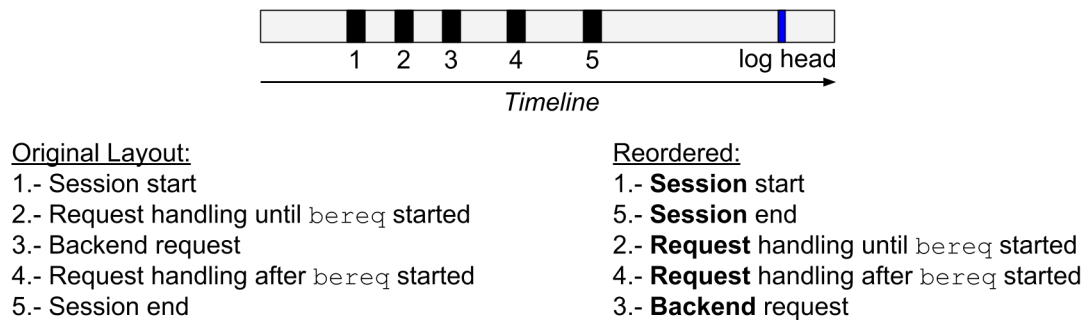


Figure 7: Log Layout Timeline

Varnish logs transactions chronologically as [Figure 7](#) shows. The `varnishlog` tool offers mechanisms to reorder transactions grouped by session, client- or backend-request. Next section explains transactions and how to reorder them.



4.3 Transactions

- One transaction is one work item in Varnish.
- Share a single Varnish Transaction ID (VXID) per types of transactions. Examples of transaction types are:
 - Session
 - Client request
 - Backend request
- Transaction reasons. Examples:
 - ESI request
 - restart
 - fetch

A transaction is a set of log lines that belongs together, e.g. a client request or a backend request. The Varnish Transaction IDs (VXIDs) are applied to lots of different kinds of work items. A unique VXID is assigned to each type of transaction. You can use the VXID when you view the log through `varnishlog`.

The default is to group the log by VXID. When viewing a log for a simple cache miss, you can see the backend request, the client request and then the session. They are displayed in the order they end. Some people find it a bit counter intuitive that the backend request is logged before the client request, but if you think about it makes sense.



4.3.1 Transaction Groups

- `varnishlog -g <session | request | vxid | raw>` groups together transactions
- Transaction groups are hierarchical
- Levels are equal to relationships (parents and children):

```
Level 1: Client request (cache miss)
Level 2: Backend request
Level 2: ESI subrequest (cache miss)
Level 3: Backend request
Level 3: Backend request (VCL restart)
Level 3: ESI subrequest (cache miss)
Level 4: Backend request
Level 2: ESI subrequest (cache hit)
```

When grouping transactions, there is a hierarchy structure showing which transaction initiated what. In client request grouping mode, the various work items are logged together with their originating client request. For example, a client request that triggers a backend request might trigger two more ESI subrequests, which in turn might trigger yet another ESI subrequest.

All these requests together with the client response are arranged in the order they are initiated. This arrangement is easier to grasp than when grouping by VXID. The [Content Composition](#) section shows how to analyze the log for Edge Side Includes (ESI) transactions.

When a subrequest occurs in the log, the subrequest tells us about the relationship to its parent request through the *Link* statement. This statement contains the VXID of the parent request. `varnishlog` indents its output based on the level of the request, making it easier to see the level of the current request.



4.3.2 Example of Transaction Grouping with *varnishlog*

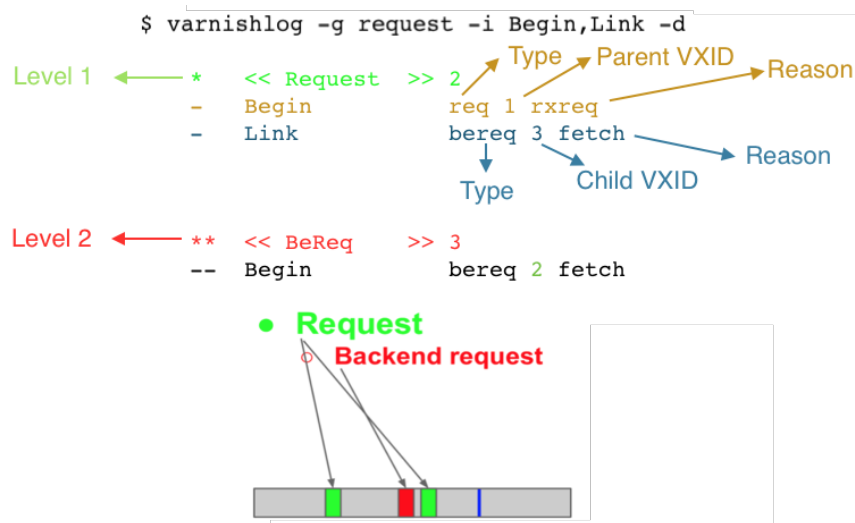


Figure 8: Example of Transaction Grouping with *varnishlog*

Figure 8 shows a client request in a *cache miss* scenario. In the figure, *varnishlog* returns records grouped by request. For simplicity, we use the *-i* option to include only the *Begin* and *Link* tags.

For more information about the format and content of all Varnish shared memory logging (VSL) tags, see the VSL man page by typing `man vsl`. The workflow of Varnish is detailed in the [VCL Basics](#) chapter.

To reproduce the example, issue `http -p hH http://localhost/`, and then the *varnishlog* command as above. The *-d* option processes all recorded entries in Varnish log. To learn more about the available *varnishlog* options, enter `varnishlog -h` or see the *varnishlog* man page.

varnishlog accepts all options that are syntactically correct. The output, however, might be different from your first interpretation. Therefore, you should make sure that your results make sense.

Options *-b* and *-c* display only transactions coming from the backend and client communication respectively. You can verify the meaning of your results by double checking the filters, and separating your results with the *-b* and *-c* options.



4.4 Query Language

- Operates on transaction groups.
- Query expression is true if it matches one or more records, false otherwise.
- Supports:
 - string matching, e.g.: `RespProtocol eq "HTTP/1.1"`
 - regex, e.g.: `ReqMethod ~ "GET|POST"`
 - integer and float matching, e.g.: `RespStatus == 200`
 - boolean operators, e.g.: `RespStatus >= 500 and RespStatus < 600`
 - parenthesis hierarchy
 - negation using `not`

Examples of Varnish log queries:

```
varnishlog -q 'RespStatus < 500'
varnishlog -g request -q 'ReqURL eq "/"'
varnishlog -g request -q 'Backend ~ default'
```

The `-q` option allows you to add a query to `varnishlog`. Think of it as a sort of select filter for `varnishlog`. It works together with the grouping so that if the query matches some part of any of the work items in the transaction group, the whole group matches and gets displayed.

Query expressions can be combined using boolean functions. There are many output control options, such as `-i taglist`. These options are output filters, they do not affect transaction matching. Output controls are applied last.

A query expression consists of record selection criteria, and optionally an operator and a value to match against the selected records:

```
<record selection criteria> <operator> <operand>
```

The `<record selection criteria>` determines what kind of records from the transaction group the expression applies to. The syntax is:

```
{level}taglist:record-prefix[field]
```

For example:

- Response time exceeds 1/2 second `Timestamp:Process[2] > 0.5`
- Client requests connection closed `ReqHeader:connection ~ close`
- ESI miss (-g request) `{3+}Begin ~ Bereq`



Taglists are not case-sensitive, but we recommend you to follow the same format as declared in `man vsl`.

The grouping and the query log processing all happens in the Varnish logging API. This means that other programs using the `varnishlog` API automatically get grouping and query language.

Tip

`man vsl-query` shows you more details about query expressions. `man vsl` lists all *taglists* and their syntax.



4.5 Exercise

- Make `varnishlog` only print client-requests where the *ReqURL* tag contains `/favicon.ico`.



4.6 varnishstat

Uptime mgt: 1+23:38:08			Hitrate n: 10 100 438			
Uptime child: 1+23:38:08			avg(n): 0.9967 0.5686 0.3870			
NAME	CURRENT	CHANGE	AVERAGE	AVG_10	AVG_100	AVG_1000
MAIN.uptime	171488	1.00	1.00	1.00	1.00	1.00
MAIN.ssess_conn	1055	7.98	.	8.35	4.49	2.11
MAIN.client_req	1055	7.98	.	8.35	4.49	2.11
MAIN.cache_hit	1052	7.98	.	8.35	4.49	2.10
MAIN.cache_miss	3	0.00	.	0.00	0.00	0.00
MAIN.backend_conn	4	0.00	.	0.00	0.00	0.01
MAIN.backend_toolate	3	0.00	.	0.00	0.00	0.01
MAIN.backend_recycle	4	0.00	.	0.00	0.00	0.01
MAIN.fetch_length	4	0.00	.	0.00	0.00	0.01
MAIN.pools	2	0.00	.	2.00	2.00	2.00
MAIN.threads	200	0.00	.	200.00	200.00	200.00
MAIN.threads_created	200	0.00	.	0.00	0.00	0.00
MAIN.n_object	1	0.00	.	1.00	0.85	0.81
MAIN.n_objectcore	3	0.00	.	3.00	2.85	2.81
MAIN.n_objecthead	4	0.00	.	4.00	3.89	3.83
MAIN.n_backend	1	0.00	.	1.00	1.00	1.00
MAIN.n_expired	2	0.00	.	2.00	1.76	1.33
MAIN.s_sess	1055	7.98	.	8.35	4.49	2.11
MAIN.s_req	1055	7.98	.	8.35	4.49	2.11
MAIN.s_fetch	3	0.00	.	0.00	0.00	0.00
MAIN.s_req_hdrbytes	122380	926.07	.	968.24	520.74	244.35
MAIN.s_resp_hdrbytes	376249	2854.04	2.00	2982.17	1602.59	751.87
MAIN.s_resp_bodybytes	3435094	25993.71	20.00	27177.59	14616.67	6858.74
MAIN.backend_req	4	0.00	.	0.00	0.00	0.01
MAIN.n_vcl	1	0.00	.	0.00	0.00	0.00
MAIN.bans	1	0.00	.	1.00	1.00	1.00
MAIN.n_gunzip	4	0.00	.	0.00	0.00	0.01
MGT.uptime	171488	1.00	1.00	1.00	1.00	1.00
SMA.s0.c_req	8	0.00	.	0.00	0.01	0.01
SMA.s0.c_bytes	15968	0.00	.	0.01	18.98	27.33
SMA.s0.c_freed	11976	0.00	.	0.00	12.17	18.56
SMA.s0.g_alloc	2	0.00	.	2.00	1.70	1.62
SMA.s0.g_bytes	3992	0.00	.	3991.87	3398.82	3235.53
SMA.s0.g_space	268431464	0.00	.	268431464.13	268432057.18	268432220.47
VBE.default(127.0.0.1,,8080).bereq_hdrbytes	630	0.00	.	0.00	0.70	1.13
VBE.default(127.0.0.1,,8080).beresp_hdrbytes	1128	0.00	.	0.00	1.34	1.93
VBE.default(127.0.0.1,,8080).beresp_bodybytes	13024	0.00	.	0.01	15.48	22.29
MAIN.cache_hit						
Cache hits:						
Count of cache hits. A cache hit indicates that an object has been delivered to a client without fetching it from a backend server.						
INFO						



Table 7: Columns displayed in central area of *varnishstat*

Column	Description
Name	The name of the counter
Current	The current value of the counter.
Change	The average per second change over the last update interval.
Average	The average value of this counter over the runtime of the Varnish daemon, or a period if the counter can't be averaged.
Avg_10	The moving average over the last 10 update intervals.
Avg_100	The moving average over the last 100 update intervals.
Avg_1000	The moving average over the last 1000 update intervals.

varnishstat looks only at counters. These counters are easily found in the SHMLOG, and are typically polled at reasonable interval to give the impression of real-time updates. Counters, unlike the rest of the log, are not directly mapped to a single request, but represent how many times a specific action has occurred since Varnish started.

varnishstat gives a good representation of the general health of Varnish. Unlike all other tools, *varnishstat* does not read log entries, but counters that Varnish updates in real-time. It can be used to determine your request rate, memory usage, thread usage, number of failed backend connections, and more. *varnishstat* gives you information just about anything that is not related to a specific request.

There are over a hundred different counters available. To increase the usefulness of *varnishstat*, only counters with a value different from 0 is shown by default.

varnishstat can be used interactively, or it can display the current values of all the counters with the `-l` option. Both methods allow you to specify specific counters using `-f field1 -f field2 ..` to limit the list.

In interactive mode, *varnishstat* has three areas. The top area shows process uptime and hitrate information. The center area shows a list of counter values. The bottom area shows the description of the currently selected counter.

Hitrate n and *avg(n)* are related, where *n* is the number intervals. *avg(n)* measures the cache hit rate within *n* intervals. The default interval time is one second. You can configure the interval time with the `-w` option.

Since there is no historical data of counters changes, *varnishstat* has to compute the average while it is running. Therefore, when you start *varnishstat*, *Hitrate* values start at 1, then they increase to 10, 100 and 1000. In the example above, the interval is one second. The hitrate average *avg(n)* show data for the last 10, 100, and 438 seconds. The average hitrate is 0.9967 (or 99.67%) for the last 10 seconds, 0.5686 for the last 100 seconds and 0.3870 for the last 438 seconds.

In the above example Varnish has served 1055 requests and is currently serving roughly 7.98 requests per second. Some counters do not have "per interval" data, but are *gauges* with values that increase and decrease. *Gauges* normally start with a *g_* prefix.



Tip

You can also see many parameters in real-time graphs with the [Varnish Administration Console \(VAC\)](#). Here are screenshots:

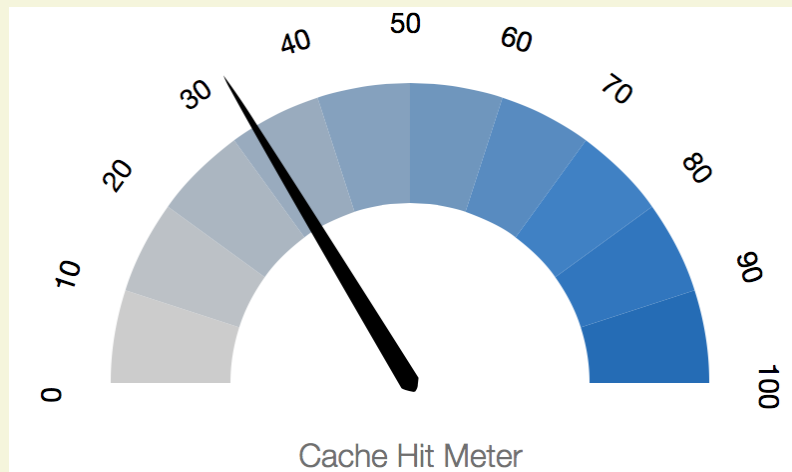


Figure 9: Cache Hit Meter

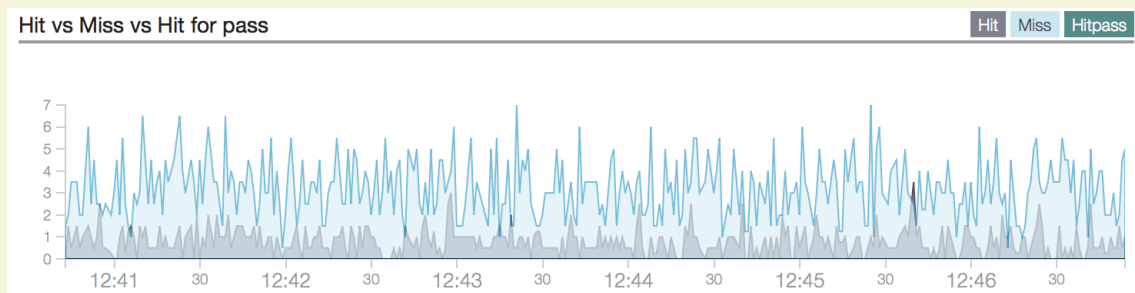


Figure 10: Hit vs Miss vs Hit for Pass

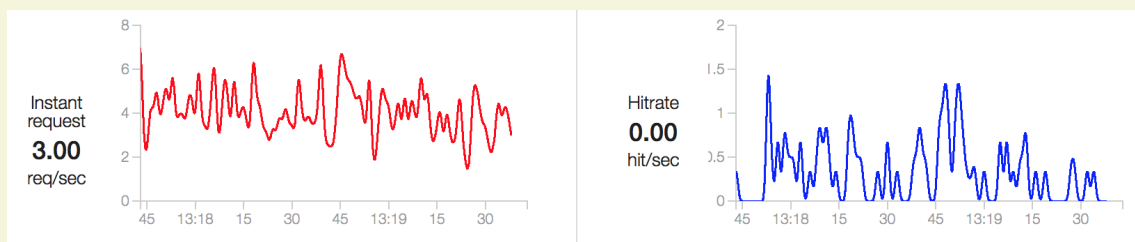


Figure 11: Req/sec, and Hit/sec



Real Time Counters	(last 10 seconds)
Cache Hit	22.50%
Cache Miss	77.50%
Cache Hit Pass	0.00%
Client Requests	4.00

Figure 12: Real Time Counters

Tip

If you need to collect statistics from more than a single Varnish server, [Varnish Custom Statistics \(VCS\)](#) allows you to do that. In addition, VCS allows you to define your metrics to collect and analyze aggregated statistics, for example:

- A/B testing
- Measuring click-through rate
- Track slow pages and cache misses
- Analyze what is "hot" right now in a news website
- Track changes in currency conversions in e-commerce
- Track changes in Stock Keeping Units (SKUs) behavior in e-commerce
- Track number of unique consumers of HLS/HDS/DASH video streams



4.6.1 Notable Counters

Table 8: Notable counters in `varnishstat`

Counter	Description
<code>MAIN.threads_limited</code>	Counts how many times <code>varnishd</code> hits the maximum allowed number of threads. The maximum number of Varnish threads is given by the parameter <code>thread_pool_max</code> . Issue the command <code>varnishadm param.show thread_pool_max</code> to see this parameter.
<code>MAIN.threads_failed</code>	Increases every time <code>pthread_create()</code> fails. You can avoid this situation by tuning the maximum number of processes available with the <code>ulimit -u</code> command. You may also look at the <code>thread-max</code> Linux parameter in <code>/proc/sys/kernel/threads-max</code> .
<code>MAIN.thread_queue_len</code>	Shows the current number of sessions waiting for a thread. This counter is first introduced in Varnish 4.
<code>MAIN.sess_queued</code>	Contains the number of sessions that are queued because there are no available threads immediately. Consider to increase the <code>thread_pool_min</code> parameter.
<code>MAIN.sess_dropped</code>	Counts how many times sessions are dropped because <code>varnishd</code> hits the maximum thread queue length. You may consider to increase the <code>thread_queue_limit</code> Varnish parameter as a solution to drop less sessions.
<code>MAIN.n_lru_nuked</code>	Number of least recently used (LRU) objects thrown out to make room for new objects. If this is zero, there is no reason to enlarge your cache. Otherwise, your cache is evicting objects due to space constraints. In this case, consider to increase the size of your cache.
<code>MAIN.n_object</code>	Number of cached objects
<code>MAIN.client_req</code>	Number of client requests
<code>MAIN.losthdr</code>	Counts HTTP header overflows produced either by client or server

Varnish provides a large number of counters for information, and debugging purposes. [Table 8](#) presents counters that are typically important. Other counters may be relevant only for Varnish developers when providing support.

Counters also provide feedback to Varnish developers on how Varnish works in production environments. This feedback in turn allows Varnish to be developed according to its real usage. Issue `varnishstat -1` to list all counters with their current values.



Tip

Remember that Varnish provides many reference manuals. To see all Varnish counter field definitions, issue `man varnish-counters`.



4.7 Exercise: Try `varnishstat` and `varnishlog` together

- Run `varnishstat` and `varnishlog` while performing a few requests.

As you are finishing up this exercise, you hopefully begin to see the usefulness of the various Varnish tools. `varnishstat` and `varnishlog` are the two most used tools, and are usually what you need for sites that are not in production yet.

The various arguments for `varnishlog` are mostly designed to help you find exactly what you want, and filter out the noise. On production traffic, the amount of log data that Varnish produces is staggering, and filtering is a requirement for using `varnishlog` effectively.



5 Tuning

This chapter is for the system administration course only

This section covers:

- Architecture
- Best practices
- Parameters

Perhaps the most important aspect of tuning Varnish is writing effective VCL code. For now, however, we will focus on tuning Varnish for your hardware, operating system and network. To be able to do that, knowledge of Varnish architecture is helpful.

It is important to know the internal architecture of Varnish for two reasons. First, the architecture is chiefly responsible for the performance, and second, it influences how you integrate Varnish in your own architecture.

There are several aspects of the design that were unique to Varnish when it was originally implemented. Truly good solutions, regardless of reusing ancient ideas or coming up with something radically different, is the aim of Varnish.



5.1 Varnish Architecture

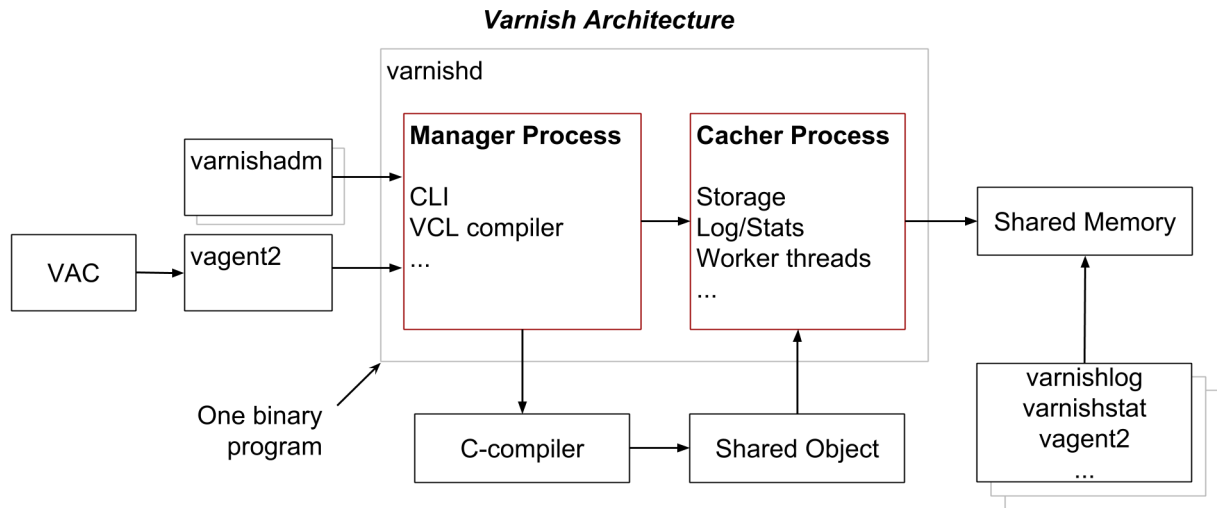


Figure 13: Varnish Architecture

Figure 13 shows a block diagram of the Varnish architecture. The diagram shows the data flow between the principal parts of Varnish.

The central block is the Varnish daemon that is contained in the `varnishd` binary program. `varnishd` creates a new child process mainly for security reasons. The parent and child processes are represented by the *Manager* and *Cacher* blocks respectively.

The Manager's command line interface (CLI) is accessible through: 1) `varnishadm` as explained in [The Management Interface `varnishadm`](#) section, 2) the Varnish Agent `vagent2`, or 2) the [Varnish Administration Console \(VAC\)](#) (via `vagent2`).

The Varnish Agent `vagent2` is an open source HTTP REST interface that exposes `varnishd` services to allow remote control and monitoring. `vagent2` offers a web UI as shown in [Figure 14](#), but you can write your own UI since `vagent2` is an open interface. Some features of the `vagent2` are:

- VCL uploading, downloading, persisting (storing to disk).
- parameter viewing, storing (not persisting yet)
- show/clear of panic messages
- start/stop/status of `varnishd`
- banning
- `varnishstat` in JON format



The screenshot shows the Varnish Agent web interface. At the top is a navigation bar with 'Varnish Agent', 'Home', 'VCL', and 'About' links, and a version number '4.0.1' on the right. The main content area is divided into several sections:

- Active VCL: boot**: A status indicator.
- Dreq/s**: A status indicator.
- Child in state running**: A status indicator.
- Parameters**: A section for configuring run-time parameters. It includes a dropdown for 'acceptor_sleep_decay' set to '0.9', with 'Save' and 'Default' buttons. Below it is a link to 'View non-default parameters'.
- Cache invalidation**: A section explaining banning. It includes a 'Ban' button and a link to 'List bans'.
- Running**: A section for managing the Varnish child process. It includes buttons for 'Start Varnish', 'Stop Varnish', 'Show panics', 'Clear panics', and 'Induce Panic'.
- Varnishtop**: A section for monitoring Varnish. It includes a 'Cache misses' dropdown, a numeric input set to '5', and an 'Update list length' button.

Figure 14: Varnish Agent's HTML interface; designed to showcase the various features of the Varnish Agent.

For more information about *vagent2* and installation instructions, please visit <https://github.com/varnish/vagent2>.

Varnish Software has a commercial offering of a fully functional web UI called **Varnish Administration Console (VAC)**. For more information about VAC, refer to the **Varnish Administration Console (VAC)** section.



5.1.1 The Parent Process: The Manager

The *Manager* process is owned by the root user, and its main functions are:

- apply configuration changes (from VCL files and parameters)
- compile VCL
- monitor Varnish
- provide a Varnish command line interface (CLI)
- initialize the *Cacher*

The *Manager* checks every few seconds whether the *Cacher* is still there. If the *Manager* does not get a reply within a given interval defined in `ping_interval`, the *Manager* kills the *Cacher* and starts it up again. This automatic restart also happens if the *Cacher* exits unexpectedly, for example, from a segmentation fault or assert error.

Automatic restart of child processes is a resilience property of Varnish. This property ensures that even if Varnish contains a critical bug that crashes the child, the child starts up again usually within a few seconds. You can toggle this property using the `auto_restart` parameter.

Note

Even if you do not perceive a lengthy service downtime, you should check whether the Varnish child is being restarted. This is important, because child restarts introduce extra loading time as `varnishd` is constantly emptying its cache. Automatic restarts are logged into `/var/log/syslog`.

To verify that the child process is not being restarted, you can also check its lifetime with the `MAIN.uptime` counter in `varnishstat`.

Varnish Software and the Varnish community at large occasionally get requests for assistance in performance tuning Varnish that turn out to be crash-issues.



5.1.2 The Child Process: The Cacher

Since the *Cacher* listens on public IP addresses and known ports, it is exposed to evil clients. Therefore, for security reasons, this child process is owned by an unprivileged user, and it has no backwards communication to its parent, the *Manager*.

The main functions of the *Cacher* are:

- listen for client requests
- manage worker threads
- store caches
- log traffic
- update counters for statistics

The *Cacher* consists of several different types of threads, including, but not limited to:

- Acceptor thread to accept new connections and delegate them.
- Worker threads - one per client request (session). It's common to use hundreds of worker threads.
- Expiry thread, to evict old content from the cache.

Varnish uses workspaces to reduce the contention between each thread when they need to acquire or modify memory. There are multiple workspaces, but the most important one is the session workspace, which is used to manipulate session data. An example is changing *www.example.com* to *example.com* before it is entered into the cache, to reduce the number of duplicates.

It is important to remember that even if you have 5MB of session workspace and are using 1000 threads, the actual memory usage is not 5GB. The virtual memory usage will indeed be 5GB, but unless you actually use the memory, this is not a problem. Your memory controller and operating system will keep track of what you actually use.

To communicate with the rest of the system, the child process uses the SHMLOG accessible from the file system. This means that if a thread needs to log something, all it has to do is to grab a lock, write to a memory area and then free the lock. In addition to that, each worker thread has a cache for log-data to reduce lock contention.

The log file is usually about 80MB, and split in two. The first part is counters, the second part is request data. To view the actual data, a number of tools exist that parses the SHMLOG. Since the log-data is not meant to be written to disk in its raw form, Varnish can afford to be very verbose. You then use one of the log-parsing tools to extract the piece of information you want -- either to store it permanently or to monitor Varnish in real-time.

If something goes wrong in the *Cacher*, it logs a detailed panic message to syslog. For testing, you can induce panic to `varnishd` by issuing the command `varnishadm debug.panic.worker` or by pressing the *Induce Panic* button in the Varnish Agent web interface.



5.1.3 VCL Compilation

The below command prints VCL code compiled to C language and exit. This is useful to check whether your VCL code compiles correctly.

```
$varnishd -C -f <filename>
```

Configuring the caching policies of Varnish is done in the Varnish Configuration Language (VCL). Your VCL is then interpreted by the *Manager* process into C, compiled by a normal C compiler – typically *gcc*, and linked into the running Varnish instance. Since the VCL compilation is done outside of the child process, there is no risk of affecting the running Varnish by accidentally loading an ill-formatted VCL.

As a result of this, changing configuration while running Varnish is very cheap. Policies of the new VCL takes effect immediately. However, objects cached with an older configuration may persist until they have no more old references or the new configuration acts on them.

A compiled VCL file is kept around until you restart Varnish completely, or until you issue `vcl.discard` from the management interface. You can only discard compiled VCL files after all references to them are gone. You can see the amount of VCL references by reading the parameter `vcl.list`.



5.2 Storage Backends

Varnish supports different methods to allocate space for the cache. You can select one method with the `-s` option of `varnishd`.

- *malloc*
- *file*
- *persistent* (deprecated)
- *Varnish Massive Storage Engine (MSE)*

Note

As a rule of thumb use: *malloc* if it fits in memory, or *file* otherwise. Expect around 1kB of overhead per object cached.

They approach the same basic problem from different angles. With the `-s malloc` method, Varnish will request the entire size of the cache with a `malloc()` (memory allocation) library call. The operating system divides the cache between memory and disk by swapping out what it can't fit in memory.

Another possibility is to use the `-s file` storage backend. This option creates a file on a filesystem to contain the entire cache. Then, the operating system maps the entire file into memory if possible.

The `-s file` storage method does not retain data when you stop or restart Varnish! For this purpose, Varnish provides a persistence option `-s persistent`. The usage of this option, however, is strongly discouraged mainly because of the consistency issues that arise with it.

The Varnish Massive Storage Engine (MSE) is an improved storage backend for Varnish Plus only. Its main improvements are decreased disk IO load and lower storage fragmentation. MSE is designed and tested with storage sizes up to 10 TB.

When choosing storage backend, use *malloc* if your cache will be contained entirely or mostly in memory. If your cache will exceed the available physical memory, you have two options: *file* or MSE. We recommend you to use MSE because it performs much better than *file* storage backend.

It is important to keep in mind that the size you specify with the `-s` option is the size for the actual cache. Varnish has an overhead on top of this for keeping track of the cache, so the actual memory footprint of Varnish will exceed what the `-s` argument specifies if the cache is full. The current estimate (subject to change on individual Varnish-versions) is that about 1kB of overhead needed for each object. For 1 million objects, that means 1GB extra memory usage.

In addition to the per-object overhead, there is also a fairly static overhead which you can calculate by starting Varnish without any objects. Typically around 100MB.



5.3 The SHared Memory Log (SHMLOG)

- Avoid I/O operations.
- Mount the shared memory log as *tmpfs*.
- *shm-log* is not persistent.

Varnish' SHared Memory LOG (SHMLOG) is used to log most data. It is sometimes called a *shm-log*, and operates on a circular buffer. The SHMLOG is 80MB large by default, which gives a certain history, but it is not persistent unless you instruct Varnish to do otherwise.

There is not much you have to do with the SHMLOG, except ensure that it does not cause I/O operations. You can avoid I/O by mounting the SHMLOG as a temporary file storage (*tmpfs*). This is typically configured in */etc/fstab*, and the *shm-log* is normally kept under */var/lib/varnish/* or equivalent locations. You need to restart *varnishd* after mounting it as *tmpfs*.

The SHMLOG is not persistent. You can issue *varnishlog -d* to see old log entries. All the content under */var/lib/varnish/* directory is safe to delete.

Warning

Some Varnish distribution setup the *file* storage backend option *-s file* by default. Those distribution set a path that puts the storage file in the same directory as the *shm-log*. We discourage this practice.



5.4 Tunable Parameters

- In the CLI:

```
param.show -l
```

- Do not fall for the copy/paste tips
- Test the parameters in CLI, then store them in the configuration file

Varnish has many different parameters which can be adjusted to make Varnish act better under specific workloads or with specific software and hardware setups. They can all be viewed with `param.show` in the management interface `varnishadm`.

You can set up parameters in two different ways. In `varnishadm`, use the command `param.set <param> <value>`. Alternatively, you can issue the command `varnishd -p param=value`.

Remember that changes made in the management interface are not persistent. Therefore, unless you store your changes in a startup script, they will be lost when Varnish restarts.


The general advice with regards to parameters is to keep it simple. Most of the defaults are optimal. If you do not have a very specific need, it is generally better to use the default values.

A few hidden debug commands exist in the CLI, which can be revealed with `help -d`. These commands are meant exclusively for development or testing, and many of them are downright dangerous. They are hidden for a reason, and the only exception is perhaps `debug.health`, which is somewhat common to use.

Tip

Parameters can also be configured via the [Varnish Administration Console \(VAC\)](#) as shown in the figure below.



 VAC

OVERVIEW


CONFIGURE

BANS

USERS

SUPPORT

MESSAGES

Signed as [vac](#)  Sign out

Parameters

CLOSE PARAMETERS


Parameter ID: 553e1b450cf2f2135d9f2c63

Name:

SAVE SET

Key	Value	<input checked="" type="checkbox"/> Validate
Parameter: <input type="text" value="key"/>	<input type="text" value="value"/>	<div>ADD</div>

Key	Value	
thread_pool_max	200	<div>REMOVE</div>

 SUGGESTED PARAMETERS

thread_pool_add_delay

thread_pool_add_threshold

thread_pool_fail_delay

thread_pool_max

USE PARAMETER

Default: 500

Unit: threads

Value: 500

Description:
The maximum number of worker threads in each pool. Do not set this higher than you have to, since excess worker threads soak up RAM and CPU and generally just get in the way of getting work done. NB: This parameter may take quite some time to take (full) effect. NB: We do not know yet if it is a good idea to change this parameter, or if the default value is even sensible. Caution is advised, and feedback is most welcome.

Figure 15: GUI to configure parameters via the *Varnish Administration Console (VAC)*.

5.5 Varnish Tuner

- Command `varnishtuner`
- Suggested values for system variables and Varnish parameters are **installation specific**
- With or without user input
- Available for Varnish Plus only

The biggest potential for improvement is outside Varnish. First and foremost in tuning the network stack and the TCP/IP connection handling.

Varnish Tuner is a program toolkit based on the experience and documentation we have built. The toolkit tries to gather as much information as possible from your installation and decides which parameters need tuning.

The tuning advice that the toolkit gives is specific to that system. The Varnish Tuner gathers information from the system it is running in. Based on that information, it suggests values for systems variables of your OS and parameters for your Varnish installation that can be beneficial to tune. Varnish Tuner includes the following information for each suggested system variable or parameter:

- current value
- suggested value
- text explaining why it is advised to be changed

`varnishtuner` requires by default user input to produce its output. If you are not sure about the requested input, you can instruct `varnishtuner` to do not suggest parameters that require user input. For this, you issue `varnishtuner -n`.

Varnish Tuner is valuable to both experts and non-experts. Varnish Tuner is available for Varnish Plus series only.

Warning

Copying Varnish Tuner suggestions to other systems might not be a good idea.



5.5.1 Varnish Tuner Persistence

The output of `varnishtuner` updates every time you introduce a new input or execute a suggested command. However, the result of the suggested commands are not necessarily persistent, which means that they do not survive a reboot or restart of Varnish Cache. To make the tuning persistent, you can add do the following:

- Specify the Varnish parameters in the configuration file.
- Specify the `sysctl` system variables in `/etc/sysctl.conf` or in `/etc/sysctl.d/varnishtuner.conf` (if `/etc/sysctl.d/` is included).

To see the usage documentation of Varnish Tuner, execute: `varnishtuner --help`.



5.5.2 Install Varnish Tuner

Below are the installation instructions for getting the tuner from our repositories. Replace the `<username>` and `<password>` with the ones of your Varnish Plus subscription. If you do not know them, please send an email to our support email to recover them.

Ubuntu Trusty 14.04

Packages in our repositories are signed and distributed via https. You need to enable https support in the package manager and install our public key first:

```
apt-get install -y apt-transport-https
curl https://<username>:<password>@repo.varnish-software.com/GPG-key.txt |
apt-key add -
```

You add the Varnish Plus repository to `/etc/apt/sources.list.d/varnish-plus.list`:

```
# Varnish Tuner
deb https://<username>:<password>@repo.varnish-software.com/ubuntu
<distribution_codename> non-free
```

Where `<distribution_codename>` is the codename of your Linux distribution, for example: `trusty`, `debian`, or `wheezy`.

Then:

```
apt-get update
apt-get install varnishtuner
```

Red Hat Enterprise Linux 6

To install Varnish Plus on RHEL6, put the following lines into `/etc/yum.repos.d/varnish-plus.repo`:

```
[varnishtuner]
name=Varnishtuner
baseurl=https://<username>:<password>@repo.varnish-software.com/redhat/ \
varnishtuner/el6
enabled=1
gpgcheck=0
```



5.6 Threading Model

- The child process runs multiple threads in two thread pools
- Worker threads are the bread and butter of the Varnish architecture
- Utility-threads

Table 9: Relevant threads in Varnish

Thread-name	Amount of threads	Task
cache-worker	1 per active connection	Handle requests
cache-main	1	Startup
ban lurker	1	Clean bans
acceptor	1	Accept new connections
epoll/kqueue	Configurable, default: 2	Manage thread pools
expire	1	Remove old content
backend poll	1 per backend poll	Health checks

The child process runs multiple threads in two thread pools. The threads of these pools are called worker threads. [Table 9](#) presents relevant threads.



5.7 Threading parameters

- Thread pools can safely be ignored
- Maximum: roughly 5000 (total)
- Start them sooner rather than later
- Maximum and minimum values are per thread pool

Table 10: Threads parameters

Parameter	Default	Description
<code>thread_pool_add_delay</code>	2 [milliseconds]	Period of time to wait for subsequent thread creation.
<code>thread_pool_destroy_delay</code>	1 second	Added time to <code>thread_pool_timeout</code> .
<code>thread_pool_fail_delay</code>	200 [milliseconds]	Period of time before retrying the creation of a thread. This after the creation of a thread failed.
<code>thread_pool_max</code>	500 [threads]	Maximum number of worker threads per pool.
<code>thread_pool_min</code>	5 [threads]	Minimum number of worker threads per pool.
<code>thread_pool_stack</code>	65536 [bytes]	Worker thread stack size.
<code>thread_pool_timeout</code>	300 [seconds]	Period of time before idle threads are destroyed.
<code>thread_pools</code>	2 [pools]	Number of worker thread pools.
<code>thread_queue_limit</code>	20 requests	Permitted queue length per thread-pool.
<code>thread_stats_rate</code>	10 [requests]	Maximum number of jobs a worker thread may handle before it is forced to dump its accumulated stats into the global counters.
<code>workspace_thread</code>	2 [kb]	Bytes of auxillary workspace per thread.

To tune Varnish, think about the expected traffic. The most important thread setting is the number of cache-worker threads. You may configure `thread_pool_min` and `thread_pool_max`. These parameters are per thread pool.

Although Varnish threading model allows you to use multiple thread pools, we recommend you to do not modify this parameter. Time and experience shows that 2 thread pools are enough. Adding more pools will not increase performance.

Note

If you run across the tuning advice that suggests to have a thread pool per CPU core, rest assured that this is old advice. Experiments and data from production environments have revealed that as long as you have two thread pools (which is the default), there is nothing to



gain by increasing the number of thread pools. Still, you may increase the number of threads per pool.

All other thread variables are not configurable.



5.7.1 Details of Threading Parameters

Most parameters can be left to the defaults with the exception is the number of threads. Varnish will use one thread for each session and the number of threads you let Varnish use is directly proportional to how many requests Varnish can serve concurrently.

Among these, `thread_pool_min` and `thread_pool_max` are the most important parameters. Values of these parameters are per thread pool. The `thread_pools` parameter is mainly used to calculate the total number of threads. For the sake of keeping things simple, the current best practice is to leave `thread_pools` at the default 2 [pools].

Varnish operates with multiple pools of threads. When a connection is accepted, the connection is delegated to one of these thread pools. Afterwards, the thread pool either delegates the connection request to an available thread, queue the request otherwise, or drop the connection if the queue is full. By default, Varnish uses 2 thread pools, and this has proven sufficient for even the most busy Varnish server.



5.7.2 Number of Threads

Varnish has the ability to spawn new worker threads on demand, and remove them once the load is reduced. This is mainly intended for traffic spikes. It's a better approach to try to always keep a few threads idle during regular traffic than it is to run on a minimum amount of threads and constantly spawn and destroy threads as demand changes. As long as you are on a 64-bit system, the cost of running a few hundred threads extra is very low.

The `thread_pool_min` parameter defines how many threads will be running for each thread pool even when there is no load. `thread_pool_max` defines the maximum amount of threads that will be used per thread pool. That means that with the minimum defaults 5 [threads] and 500 [threads] of minimum and maximums threads per pool respectively, you have:

- at least 5 [threads] * 2 [pools] worker threads at any given time
- no more than 500 [threads] * 2 [pools] worker threads ever

We rarely recommend running Varnish with more than 5000 threads. If you seem to need more than 5000 threads, it is very likely that there is something wrong in your setup. Therefore, you should investigate elsewhere before you increase the maximum value.

For minimum, it's common to operate with 500 to 1000 threads minimum (total). You can observe if those values are enough by looking at `MAIN.sess_queued` through `varnishstat`. Look at the counter over time, because it is fairly static right after startup.

Warning

New threads use preallocated workspace. If threads have not enough workspace, the child process is unable to process the task and it terminates. The workspace needed depends on the task that the thread handles. This is normally defined in your VCL. To avoid that the child terminates, evaluate your VCL code and consider to increase the `workspace_client` or `workspace_backend` parameter.



5.7.3 Timing Thread Growth

Varnish can use several thousand threads, and has had this capability from the very beginning. However, not all operating system kernels were prepared to deal with this capability. Therefore the parameter `thread_pool_add_delay` was added to ensure that there is a small delay between each thread that spawns. As operating systems have matured, this has become less important and the default value of `thread_pool_add_delay` has been reduced dramatically, from 20 ms to 2 ms.

There are a few, less important parameters related to thread timing. The `thread_pool_timeout` is how long a thread is kept around when there is no work for it before it is removed. This only applies if you have more threads than the minimum, and is rarely changed.

Another less important parameter is the `thread_pool_fail_delay`. After the operating system fails to create a new thread, `thread_pool_fail_delay` defines how long to wait for a re-trial.



5.8 System Parameters

As Varnish has matured, fewer and fewer parameters require tuning. The `workspace_client` and `workspace_backend` are parameters that could still be relevant.

- `workspace_client` – incoming HTTP header workspace from the client
- `workspace_backend` – bytes of HTTP protocol workspace for backend HTTP req/resp
- ESI typically requires exponential growth
- Remember: it is virtual, not physical memory

Workspaces are some of the things you can change with parameters. Sometimes you may have to increase them to avoid running out of workspace.

The `workspace_client` parameter states how much memory can be allocated for each HTTP session. This space is used for tasks like string manipulation of incoming headers. The `workspace_backend` parameter indicates how much memory can be allocated to modify objects returned from the backend. After an object is modified, its exact size is allocated and the object is stored read-only.

As most of the parameters can be left unchanged, we will not go through all of them. You can take a look at the list of parameter by issuing `varnishadm param.show -1` to get information about what they can do.



5.9 Timers

Table 11: Timers

Parameter	Default	Description	Scope
<code>connect_timeout</code>	3.5 [s]	OS/network latency	Backend
<code>first_byte_timeout</code>	60 [s]	Web page generation	Backend
<code>between_bytes_timeout</code>	60 [s]	Hiccoughs	Backend
<code>send_timeout</code>	600 [seconds]	Client-in-tunnel	Client
<code>timeout_idle</code>	5 [seconds]	keep-alive timeout	Client
<code>timeout_req</code>	2 [seconds]	deadline to receive a complete request header	Client
<code>cli_timeout</code>	60 [seconds]	Management thread->child	Management

The timeout-parameters are generally set to pretty good defaults, but you might have to adjust them for unusual applications. The default value of `connect_timeout` is 3.5 [s]. This value is more than enough when having the Varnish server and the backend in the same server room. Consider to increase the `connect_timeout` value if your Varnish server and backend have a higher network latency.

Keep in mind that the session timeout affects how long sessions are kept around, which in turn affects file descriptors left open. It is not wise to increase the session timeout without taking this into consideration.

The `cli_timeout` is how long the management thread waits for the worker thread to reply before it assumes it is dead, kills it and starts it back up. The default value seems to do the trick for most users today.

Warning

If `connect_timeout` is set too high, it does not let Varnish handle errors gracefully.

Note

Another use-case for increasing `connect_timeout` occurs when virtual machines are involved, as they can increase the connection time significantly.



Tip

More information in
<https://www.varnish-software.com/blog/understanding-timeouts-varnish-cache>.



5.10 Exercise: Tune `first_byte_timeout`

- Run `varnishstat` and `varnishlog` while performing a few requests.
 - See, analyze and understand how counters in `varnishstat` and parameters in `varnishlog` change.
- Create a small CGI script under `/usr/lib/cgi-bin/` containing:

```
#!/bin/sh
sleep 5
echo "Content-type: text/plain"
echo "Cache-control: max-age=0"
echo
echo "Hello world"
date
```

1. Make it executable.
2. Test that it works without involving of Varnish.
3. Test it through Varnish.
4. Set `first_byte_timeout` to 2 seconds.
5. Check how Varnish times out the request to the backend.

Tip

You may need to enable the `cgi` module in `apache`. One way to do that, is by issuing the commands: `a2enmod cgi`, and then `service apache2 restart`.



5.11 Exercise: Configure threading

While performing this exercise, watch the *MAIN.threads* counter in `varnishstat` to know how many threads are running.

- Change the `thread_pool_min` and `thread_pool_max` parameters to get 100 threads running at any given time, but never more than 400.
- Make the changes work across restarts of Varnish.

Extra: Experiment with `thread_pool_add_delay` and `thread_pool_timeout` while watching `varnishstat` to see how thread creation and destruction is affected. Does `thread_pool_timeout` affect already running threads?

You can also try changing the `thread_pool_stack` variable to a lower value. This will only affect new threads, but try to find out how low you can set it, and what happens if it's too low.

Note

It's not common to modify `thread_pool_stack`, `thread_pool_add_delay` or `thread_pool_timeout`. These assignments are for educational purposes, and not intended as an encouragement to change the values.



6 HTTP

This chapter is for the webdeveloper course only

This chapter covers:

- Protocol basics
- Requests and responses
- HTTP request/response control flow
- Statelessness and idempotence
- Cache related headers

HTTP is at the heart of Varnish, or rather the model HTTP represents.

This chapter will cover the basics of HTTP as a protocol, how it's used in the wild, and delve into caching as it applies to HTTP.



6.1 Protocol basics

- Hyper-Text Transfer Protocol, HTTP, is at the core of the web
- Specified by the IETF, the latest version (HTTP/1.1) is available from <http://tools.ietf.org/html/rfc2616>
- A request consists of a request method, headers and an optional request body.
- A response consists of a response status, headers and an optional response body.
- Multiple requests can be sent over a single connection, in serial.
- Clients will open multiple connections to fetch resources in parallel.

HTTP is a networking protocol for distributed systems. It is the foundation of data communication for the Web. The development of this standard is done by the IETF and the W3C. The latest version of the standard is HTTP/1.1. In 2014, new RFCs have been published to clarify (and obsolete) 2616: RFCs 7230 to 7235.

A new version of HTTP called HTTP/2 has been released, but RFCs haven't been published yet. Basically HTTP/2 is a radically different protocol but shares the same goals.



6.2 Requests

- Standard request methods are: *GET*, *POST*, *HEAD*, *OPTIONS*, *PUT*, *DELETE*, *TRACE*, or *CONNECT*.
- This is followed by a URI, e.g: */img/image.png* or */index.html*
- Usually followed by the HTTP version
- A new-line (CRLF), followed by an arbitrary amount of CRLF-separated headers (*Accept-Language*, *Cookie*, *Host*, *User-Agent*, etc).
- A single empty line, ending in CRLF.
- An optional message body, depending on the request method.

Each request has the same, strict and fairly simple pattern. A request method informs the web server what sort of request this is: Is the client trying to fetch a resource (*GET*), or update some data(*POST*)? Or just get the headers of a resource (*HEAD*)?

There are strict rules that apply to the request methods. For instance, a *GET* request should not contain a request body, but a *POST* request can.

Similarly, a web server can not attach a request body to a response to a *HEAD* body.



6.3 Request example

```
GET / HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; fr; rv:1.9.2.16) \
Gecko/20110319 Firefox/3.6.16
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cache-Control: max-age=0
```

The above is a typical HTTP *GET* request for the `/` resource.

Note that the `Host`-header contains the hostname as seen by the browser. The above request was generated by entering <http://localhost/> in the browser. The browser automatically adds a number of headers. Some of these will vary depending on language settings, others will vary depending on whether the client has a cached copy of the page already, or if the client is doing a refresh or forced refresh.

Whether the server honors these headers will depend on both the server in question and the specific header.

The following is an example of an HTTP request using the *POST* method, which includes a request body:

```
POST /accounts/ServiceLoginAuth HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; fr; rv:1.9.2.16) \
Gecko/20110319 Firefox/3.6.16
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: https://www.google.com/accounts/ServiceLogin
Cookie: GoogleAccountsLocale_session=en;[...]
Content-Type: application/x-www-form-urlencoded
Content-Length: 288

ltmpl=default[...]&signIn=Sign+in&asts=
```



6.4 Response

```
HTTP/1.1 200 OK
Cache-Control: max-age=150
Content-Length: 150

[data]
```

- An HTTP response contains the HTTP versions, a status code (e.g: 200) and a reason (e.g: OK)
- CRLF as line separator
- A number of headers
- Headers are terminated with a blank line
- Optional response body

The HTTP response is similar to the request itself. The response code informs the browser both whether the request succeeded and what type of response this is. The response message is a text-representation of the same information, and is often ignored by the browser itself.

Examples of status codes are *200 OK*, *404 File Not Found*, *304 Not Modified* and so fort. They are all defined in the HTTP standard, and grouped into the following categories:

- 1xx: Informational - Request received, continuing process
- 2xx: Success - The action was successfully received, understood, and accepted
- 3xx: Redirection - Further action must be taken in order to complete the request
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled
- 5xx: Server Error - The server failed to fulfill an apparently valid request

Note

The main difference between a request and a response (besides the semantics) is the start line. Both share the same syntax for headers and the body, but some headers are request- or response-specific.



6.5 Response example

```
HTTP/1.1 200 OK
Server: Apache/2.2.14 (Ubuntu)
X-Powered-By: PHP/5.3.2-1ubuntu4.7
Cache-Control: public, max-age=86400
Last-Modified: Mon, 04 Apr 2011 04:13:41 +0000
Expires: Sun, 11 Mar 1984 12:00:00 GMT
Vary: Cookie,Accept-Encoding
ETag: "1301890421"
Content-Type: text/html; charset=utf-8
Content-Length: 23562
Date: Mon, 04 Apr 2011 09:02:26 GMT
X-Varnish: 1886109724 1886107902
Age: 17324
Via: 1.1 varnish
Connection: keep-alive

[data]
```



6.6 HTTP request/response control flow

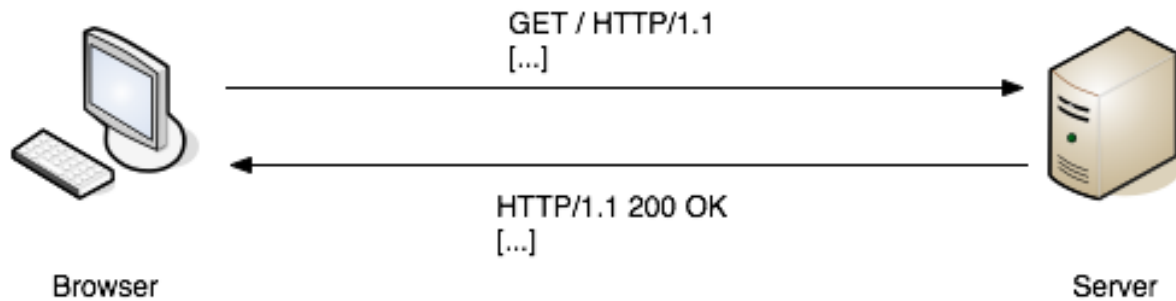


Figure 16: HTTP request/response control flow diagram

The client sends an HTTP request to the server which returns an HTTP response with the message body.



6.7 Statelessness and idempotence

statelessness

HTTP is by definition a stateless protocol which means that in theory your browser has to reconnect to the server for every request. In practice there is a header called *Keep-Alive* you may use if you want to keep the connection open between the client (your browser) and the server.

idempotence

Idempotence means that an operation can be applied multiple times without changing the result. *GET* and *PUT* HTTP request are expected to be idempotent whereas *POST* requests are not. In other words, you can not cache *POST* HTTP responses.

For more discussion about idempotence <http://queue.acm.org/detail.cfm?id=2187821>.



6.8 Cache related headers

HTTP provides a list of headers dedicated to page caching and cache invalidation. The most important ones are:

- Expires
- Cache-Control
- Etag
- Last-Modified
- If-Modified-Since
- If-None-Match
- Vary
- Age



6.9 Exercise: Test various Cache headers

Before we talk about all the various cache headers and cache mechanisms, we will use *httpheadersexample.php* to experiment and get a sense of what it's all about.

Try both clicking the links twice, hitting refresh and forced refresh (usually done by hitting control-F5, depending on browser).

- Try out the Expires-header and see how the browser and Varnish behave.
- What happens when both Expires and Cache-Control is present?
- **Test the *If-Modified-Since* request too. Does the browser issue a** request to Varnish? If the item was in cache, does Varnish query the web-server?
- Try the *Vary*-test by using two different browsers at the same time.

When performing this exercise, try to see if you can spot the patterns. There are many levels of cache on the Web, and you have to think about more than just Varnish.

If it hasn't already, it's likely that browser cache will confuse you at least a few times through this course. When that happens, pull up varnishlog or another browser.



6.10 Expires

The *Expires* **response** header field gives the date/time after which the response is considered stale. A stale cache item will not be returned by any cache (proxy cache or client cache).

The syntax for this header is:

```
Expires: GMT formatted date
```

It is recommended not to define *Expires* too far in the future. Setting it to 1 year is usually enough.

Using *Expires* does not prevent the cached resource to be updated. If a resource is updated changing its name (by using a version number for instance) is possible.

Expires works best for any file that is part of your design like JavaScripts stylesheets or images.



6.11 Cache-Control

The *Cache-Control* header field specifies directives that **must** be applied by all caching mechanisms (from proxy cache to browser cache). *Cache-Control* accepts the following arguments (only the most relevant are described):

- *public*: The response may be cached by any cache.
- *no-store*: The response body **must not** be stored by any cache mechanism;
- *no-cache*: Authorizes a cache mechanism to store the response in its cache but it **must not** reuse it without validating it with the origin server first. In order to avoid any confusion with this argument think of it as a "store-but-do-no-serve-from-cache-without-revalidation" instruction.
- *max-age*: Specifies the period in seconds during which the cache will be considered fresh;
- *s-maxage*: Like *max-age* but it applies only to public caches;
- *must-revalidate*: Indicates that a stale cache item can not be serviced without revalidation with the origin server first;

Table 12: Cache-control argument for each context

Argument	Request	Response
<i>no-cache</i>	X	X
<i>no-store</i>	X	X
<i>max-age</i>	X	X
<i>s-maxage</i>		X
<i>max-stale</i>	X	
<i>min-fresh</i>	X	
<i>no-transform</i>	X	X
<i>only-if-cached</i>	X	
<i>public</i>		X
<i>private</i>		X
<i>must-revalidate</i>		X
<i>proxy-revalidate</i>		X

Unlike *Expires*, *Cache-Control* is both a **request** and a **response** header. Table 12 summarizes the arguments you may use for each context.

Example of a *Cache-Control* header:

```
Cache-Control: public, must-revalidate, max-age=2592000
```



Note

As you might have noticed *Expires* and *Cache-Control* do more or less the same job, *Cache-Control* gives you more control though. There is a significant difference between these two headers:

- *Cache-Control* uses relative times in seconds, cf (s)max-age
- *Expires* always returns an absolute date

Note

Cache-Control **always** overrides Expires.

Note

By default, Varnish does not care about the Cache-Control request header. If you want to let users update the cache via a force refresh you need to do it yourself.



6.12 Last-Modified

The *Last-Modified* **response** header field indicates the date and time at which the origin server believes the variant was last modified. This headers may be used in conjunction with *If-Modified-Since* and *If-None-Match*.

Example of a *Last-Modified* header:

```
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
```



6.13 If-Modified-Since

The *If-Modified-Since* **request** header field is used with a method to make it conditional:

- **if** the requested variant has not been modified since the time specified in this field, an entity will not be returned from the server;
- **instead**, a 304 (not modified) response will be returned without any message-body.

Example of an *If-Modified-Since* header:

```
If-Modified-Since: Wed, 01 Sep 2004 13:24:52 GMT
```

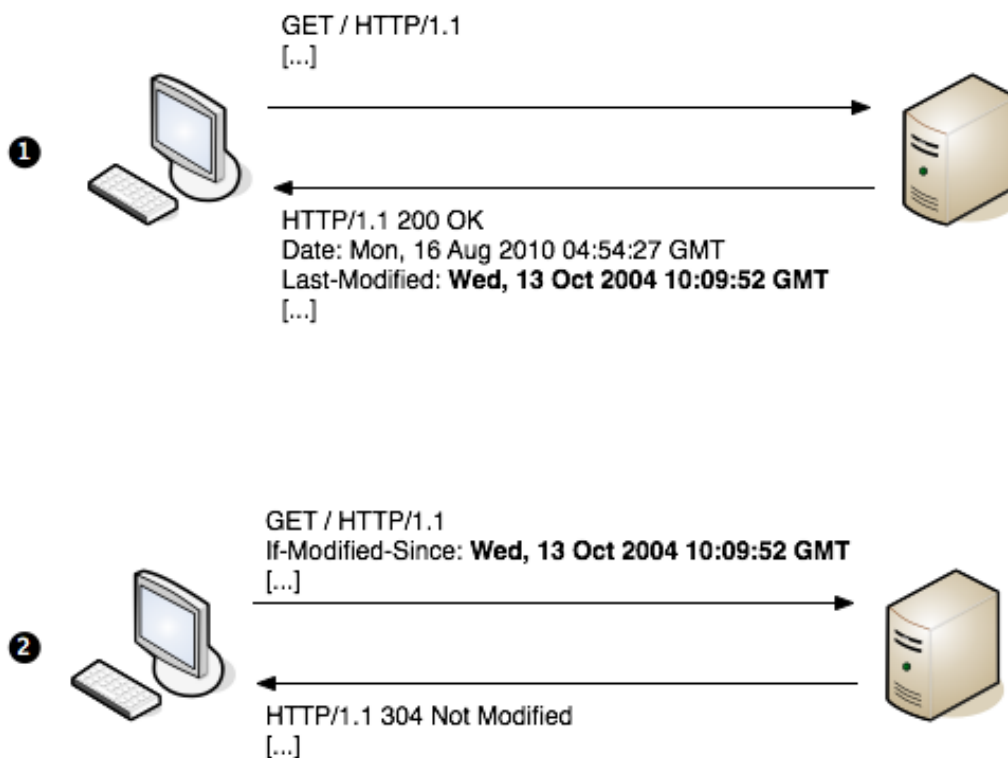


Figure 17: *If-Modified-Since* control flow diagram.



6.14 If-None-Match

The *If-None-Match* **request** header field is used with a method to make it conditional.

A client that has one or more entities previously obtained from the resource can verify that none of those entities is current by including a list of their associated entity tags in the *If-None-Match* header field.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead. It is also used to prevent a method (e.g. PUT) from inadvertently modifying an existing resource when the client believes that the resource does not exist.

Example of an *If-None-Match* header:

```
If-None-Match: "1edec-3e3073913b100"
```

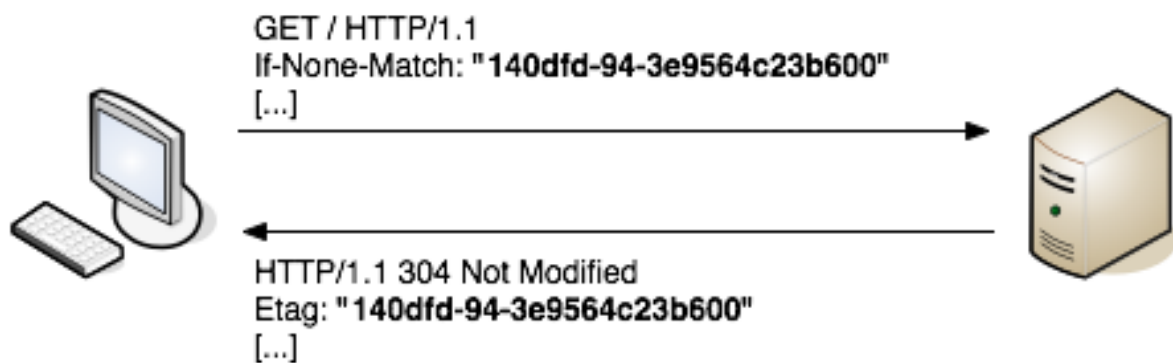


Figure 18: *If-None-Match* control diagram.



6.15 Etag

The *ETag* **response** header field provides the current value of the entity tag for the requested variant. The idea behind *Etag* is to provide a unique value for a resource's contents.

Example of an *Etag* header:

```
Etag: "1edec-3e3073913b100"
```



6.16 Pragma

The *Pragma request* header is a legacy header and should no longer be used. Some applications still send headers like `Pragma: no-cache` but this is for backwards compatibility reasons **only**.

Any proxy cache should treat `Pragma: no-cache` as `Cache-Control: no-cache`, and should not be seen as a reliable header especially when used as a response header.



6.17 Vary

The **Vary response** header indicates the response returned by the origin server may vary depending on headers received in the request.

The most common usage of *Vary* is to use `Vary: Accept-Encoding`, which tells caches (Varnish included) that the content might look different depending on the *Accept-Encoding*-header the client sends. In other words: The page can be delivered compressed or uncompressed depending on the client.

The *Vary*-header is one of the trickiest headers to deal with for a cache. A cache, like Varnish, does not necessarily understand the semantics of a header, or what part triggers different variants of a page.

As a result, using `Vary: User-Agent` for instance tells a cache that for ANY change in the *User-Agent*-header, the content *might* look different. Since there are probably thousands of *User-Agent* strings out there, this means you will drastically reduce the efficiency of any cache method.

An other example is using `Vary: Cookie` which is actually not a bad idea. Unfortunately, you can't issue `Vary: Cookie` (but only `THESE cookies: ...`). And since a client will send you a great deal of cookies, this means that just using `Vary: Cookie` is not necessarily sufficient. We will discuss this further in the Content Composition chapter.

Note

Varnish can handle *Accept-Encoding* and `Vary: Accept-Encoding`, because Varnish has support for gzip compression.



6.18 Age

- A cache server can send an additional response header, *Age*, to indicate the age of the response.
- Varnish (and other caches) does this.
- Browsers (and Varnish) will use the Age-header to determine how long to cache.
- E.g: for a *max-age*-based equation: $\text{cache duration} = \text{max-age} - \text{Age}$
- If you allow Varnish to cache for a long time, the *Age*-header could effectively disallow client-side caches.

Consider what happens if you let Varnish cache content for a week, because you can easily invalidate the cache Varnish keeps. If you do not change the *Age*-header, Varnish will happily inform clients that the content is, for example, two days old, and that the maximum age should be no more than fifteen minutes.

Browsers will obey this. They will use the reply, but they will also realize that it has exceeded its max-age, so they will not cache it.

Varnish will do the same, if your web-server emits an *Age*-header (or if you put one Varnish-server in front of another).

We will see in later chapters how we can handle this in Varnish.



6.19 Header availability summary

The table below lists HTTP headers seen above and whether they are a request header or a response one.

Table 13: Header availability summary

Header	Request	Response
Expires		X
Cache-Control	X	X
Last-Modified		X
If-Modified-Since	X	
If-None-Match	X	
Etag		X
Pragma	X	X
Vary		X
Age		X



6.20 Cache-hit and misses

cache-hit

There is a cache-hit when Varnish returns a page from its cache instead of forwarding the request to the origin server.

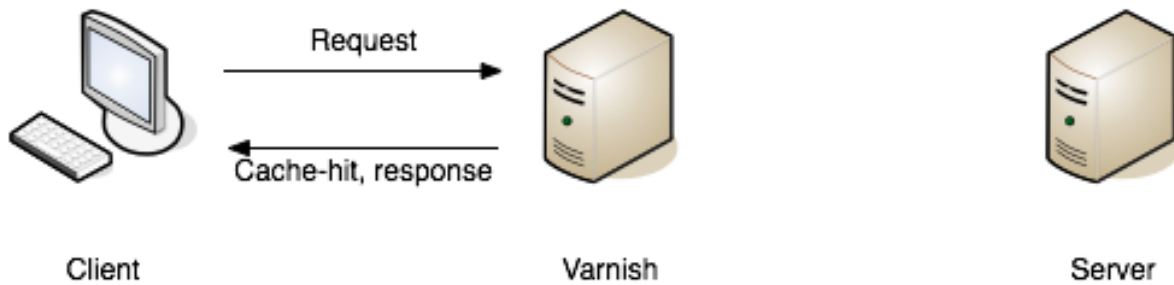


Figure 19: Cache-hit control flow diagram

cache-miss

There is a cache-miss when Varnish has to forward the request to the origin server so the page can be serviced.

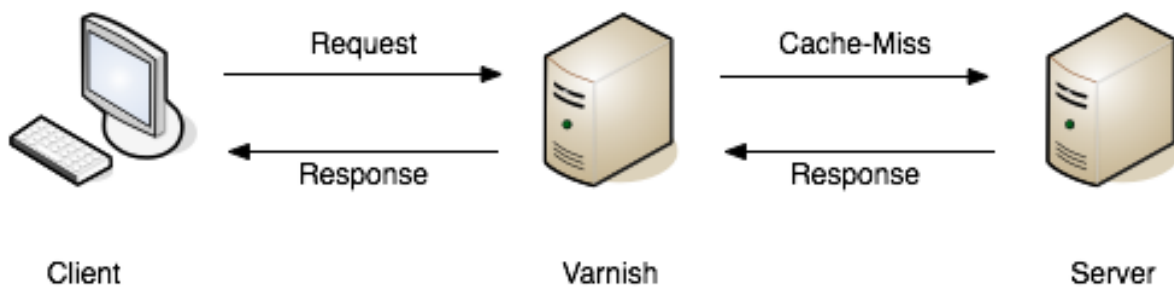


Figure 20: Cache-miss control flow diagram



6.21 Exercise: Use *article.php* to test *Age*

1. Modify the *article.php*-script to send an *Age* header that says *30* and *Cache-Control: max-age=60*.
2. Watch *varnishlog*.
3. Send a request to Varnish for *article.php*. See what *Age*-Header Varnish replies with.
4. Is the *Age*-header an accurate method to determine if Varnish made a cache hit or not?
5. How long does Varnish cache the reply? How long would a browser cache it?

Also consider how you would avoid issues like this to begin with. We do not yet know how to modify Varnish' response headers, but hopefully you will understand why you may need to do that.

Varnish is not the only part of your web-stack that parses and honors cache-related headers. The primary consumer of such headers are the web browsers, and there might also be other caches along the way which you do not control, like a company-wide proxy server.

By using *s-maxage* instead of *max-age* we limit the number of consumers to cache servers, but even *s-maxage* will be used by caching proxies which you do not control.

In the next few chapters, you will learn how to modify the response headers Varnish sends. That way, your web-server can emit response headers that are only seen and used by Varnish.



7 VCL Basics

- The Varnish Configuration Language (VCL) is a domain-specific language
- VCL as a state machine
- VCL subroutines
- Built-in subroutines
- Available functions, legal return actions and variables.

The Varnish Configuration Language (VCL) is a domain-specific language designed to describe request handling and document caching policies for Varnish Cache. When a new configuration is loaded, the `varnishd` manager process translates the VCL code to C and compiles it to a shared object. This shared object is then loaded into the cacher process.

This chapter focuses on the most important tasks to write effective VCL code. For this, you will learn the basic syntax of VCL, and the most important VCL built-in subroutines: `VCL_recv` and `VCL_backend_fetch`. All other built-in subroutines are taught in the next chapter.

Tip

Remember that Varnish has many reference manuals. For more details about VCL, check its manual page by issuing `man vcl`.



7.1 Varnish Finite State Machine



Figure 21: Simplified Version of the Varnish Finite State Machine



VCL is also often described as a finite state machine. Each state has available certain parameters that you can use in your VCL code. For example: response HTTP headers are only available after `vcl_backend_fetch` state.

[Figure 21](#) shows a simplified version of the Varnish finite state machine. This version shows by no means all possible transitions, but only a traditional set of them. [Figure 22](#) shows a detailed version of the state machine for the frontend worker as a request flow diagram. A detailed version of the request flow diagram for the backend worker is in the [VCL - `vcl_backend_fetch` and `vcl_backend_response`](#) section.

States in VCL are conceptualized as subroutines, with the exception of the *waiting* state described in [Waiting State](#)

Subroutines in VCL take neither arguments nor return values. Each subroutine terminates by calling `return (action)`, where `action` is a keyword that indicates the desired outcome. Subroutines may inspect and manipulate HTTP headers and various other aspects of each request. Subroutines instruct how requests are handled.

Subroutine example:

```
sub pipe_if_local {
    if (client.ip ~ local) {
        return (pipe);
    }
}
```

To call a subroutine, use the `call` instruction followed by the subroutine's name:

```
call pipe_if_local;
```

Varnish has built-in subroutines that are hook into the Varnish workflow. These built-in subroutines are all named `vcl_*`. Your own subroutines cannot start their name with `vcl_`.



7.1.1 *Waiting State*

- Designed to improve response performance

The *waiting* state is reached when a request *n* arrives while a previous identical request 0 is being handled by the backend. In this case, request 0 is set as *busy* and all subsequent requests *n* are queued in a waiting list. If the fetched object from request 0 is cacheable, all *n* requests in the waiting list call the lookup operation again. This retry will hopefully hit the desired object in cache. In this way, only one request is sent to the backend.

The *waiting* state is designed to improve response performance. However, a counterproductive scenario, namely *request serialization*, may occur if the fetched object is uncacheable, and so is recursively the next request in the waiting list. This situation forces every single request in the waiting list to be sent to the backend in a serial manner. Serialized requests should be avoided because their performance is normally poorer than sending multiple requests in parallel. The built-in `vcl_backend_response` subroutine avoids *request serialization*.



7.2 Detailed Varnish Request Flow for the Client Worker Thread

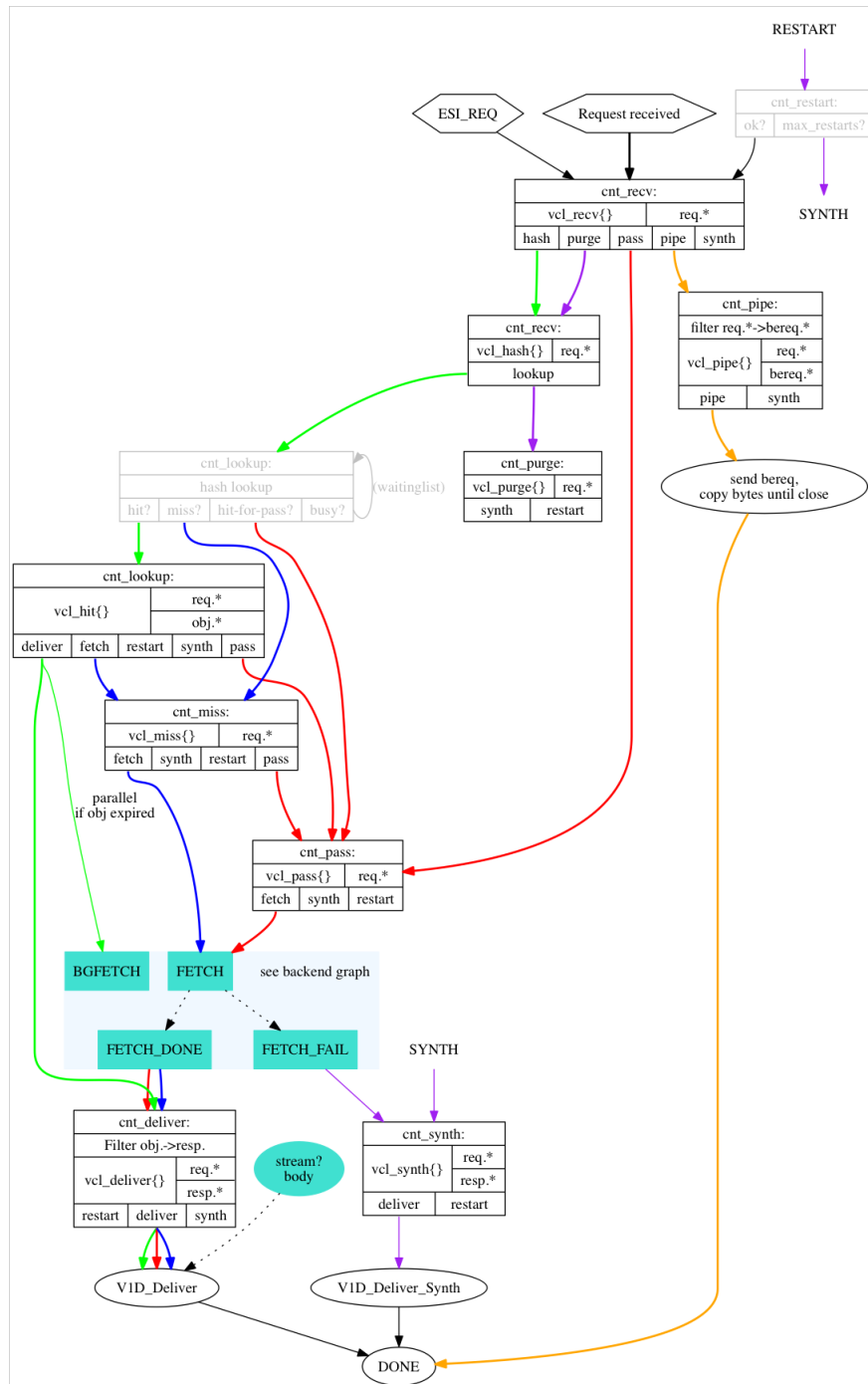


Figure 22: Detailed Varnish Request Flow for the Client Worker Thread

[Figure 22](#) shows the detailed request flow diagram of the backend worker. This diagram details the grayed box in [Figure 21](#).



7.3 The VCL Finite State Machine

- Each request is processed separately
- Each request is independent from others at any given time
- States are related, but isolated
- `return(action);` exits one state and instructs Varnish to proceed to the next state
- Built-in VCL code is always present and appended below your own VCL

Before we begin looking at VCL code, it's worth trying to understand the fundamental concepts behind VCL. When Varnish processes a request, it starts by parsing the request itself. Next, Varnish separates the request method from headers, verifying that it's a valid HTTP request and so on.

When the basic parsing has completed, the very first policies are checked to make decisions. Policies are a set of rules that the VCL code uses to make a decision. Policies help to answer questions such as: should Varnish even attempt to find the requested resource in the cache? In this example, the policies are in the `vcl_recv` subroutine.



7.4 VCL Syntax

- `//`, `#` and `/* foo */` for comments
- Subroutines are declared with the `sub` keyword
- No loops, state-limited variables
- Terminating statements with a keyword for next action as argument of the `return()` function, i.e.: `return(action)`
- Domain-specific
- Add as little or as much as you want

Starting with Varnish 4.0, each VCL file must start by declaring its version with a special `vcl 4.0;` marker at the top of the file. If you have worked with a programming language or two before, the basic syntax of Varnish should be reasonably straightforward. VCL is inspired mainly by C and Perl. Blocks are delimited by curly braces, statements end with semicolons, and comments may be written as in C, C++ or Perl according to your own preferences.

Subroutines in VCL neither take arguments, nor return values. Subroutines in VCL can exchange data only through HTTP headers.

VCL has terminating statements, not traditional return values. Subroutines end execution when a `return(*action*)` statement is made. The *action* tells Varnish what to do next. For example, "look this up in cache", "do not look this up in the cache", or "generate an error message". To check which actions are available for the built-in subroutines, you can look at Figure # or see the manual page of VCL.

Warning

If you define your own subroutine and call it from one of the built-in subroutines, executing `return(foo)` does not return execution from your custom subroutine to the default function, but returns execution from VCL to Varnish.



7.5 VCL built-in functions

List of functions and their arguments:

- `regsub(str, regex, sub)`
- `regsuball(str, regex, sub)`
- `ban(exp)`
- `return(action)`
- `hash_data(input)`
- `call subroutine`
- `new()`
- `synthetic(str)`
- `set()`
- `unset()`

All functions are available in all subroutines, except the listed in the table below.

Table 14: Specific Function Availability

Function	Subroutines
<code>hash_data</code>	<code>vcl_hash</code>
<code>new</code>	<code>vcl_init</code>
<code>synthetic</code>	<code>vcl_synth</code> , <code>vcl_backend_error</code>

VCL offers a handful of simple to use built-in functions that allow you to modify strings, add bans, restart the VCL state engine and return control from the VCL Run Time (VRT) environment to Varnish. This book describes the most important functions in later sections, so the description at this point is brief.

`regsub()` and `regsuball()` have the same syntax and does the almost same thing: They both take a string `str` as input, search it with a regular-expression `regex` and replace it with another string. The difference between `regsub()` and `regsuball()` is that the latter changes all occurrences while the former only affects the first match.

The `ban()` function invalidates all objects in cache that match the expression `exp` with the ban mechanism. *banning* and *purging* in detailed in the [Cache Invalidation](#) chapter.



7.6 Legal Return Actions

Table 15: VCL built-in subroutines and their legal returns at the frontend (client) side

subroutine	scope	deliver	fetch	restart	hash	pass	pipe	synth	purge	lookup
<i>vcl_deliver</i>	client	x		x				x		
<i>vcl_hash</i>	client									x
<i>vcl_hit</i>	client	x	x	x		x		x		
<i>vcl_miss</i>	client		x	x		x		x		
<i>vcl_pass</i>	client		x	x				x		
<i>vcl_pipe</i>	client						x	x		
<i>vcl_purge</i>	client			x				x		
<i>vcl_recv</i>	client				x	x	x	x	x	
<i>vcl_synth</i>	client	x		x						

Table 16: VCL built-in subroutines and their legal returns at the backend side, *vcl.load*, and *vcl.discard*

subroutine	scope	abandon	retry	ok	fail
<i>vcl_backend_error</i>	backend		x		
<i>vcl_backend_fetch</i>	backend	x			
<i>vcl_backend_response</i>	backend	x	x		
<i>vcl_init</i>	vcl.load			x	x
<i>vcl_fini</i>	vcl.discard			x	

The table above shows the VCL built-in subroutines and their legal returns. `return()` is a built-in function that ends execution of the current VCL subroutine, and continue to the next `action` step in the request handling state machine. Legal return actions are: *lookup*, *synth*, *purge*, *pass*, *pipe*, *fetch*, *deliver*, *hash*, *restart*, *retry*, and *abandon*.

Note

Varnish 4 defines `purge` as a return action. This is contrary to Varnish 3, where `purge` is a function.



7.7 Variables in VCL subroutines

Table 17: Variable Availability in VCL subroutines

subroutine	req.	bereq.	beresp.	obj.	resp.
<i>vcl_backend_fetch</i>		R/W			
<i>vcl_backend_response</i>		R/W	R/W		
<i>vcl_backend_error</i>		R/W	R/W		
<i>vcl_recv</i>	R/W				
<i>vcl_pipe</i>		R/W			
<i>vcl_pass</i>	R/W				
<i>vcl_hash</i>	R/W				
<i>vcl_purge</i>	R/W				
<i>vcl_miss</i>	R/W				
<i>vcl_hit</i>	R/W			R	
<i>vcl_deliver</i>	R/W			R	R/W
<i>vcl_synth</i>	R/W				R/W

The *State* column lists the different states in a request work-flow. States are handled by subroutines, which have a leading `vcl_` prefix name.

The *Variables* columns list the prefix of the available variables. Most but not all of the prefixed variables are readable (R) or writable (W) at the given state. To have a detailed availability of each variable, refer to the VCL man page by typing: `man vcl`.

Table 17 shows the availability of variables in different states of the Varnish finite state machine. In addition to the variable prefixes in Table 17, there are other three variables prefixes; `client.*`, `server.*`, and `storage.*`, which are accessible from all subroutines at the frontend (client) side. Another variable is `now`, which is accessible from all subroutines.

These additional prefixes and variable are practically accessible everywhere.

Remember that changes made to `beresp.` variables are stored in `obj.` afterwards. And the `resp.` variables are copies of what is about to be returned to the client. The values of `resp.` variables come possibly from `obj.`

A change to `beresp.` variables, in other words, affects `obj.` and `resp.` variables. Similar semantics apply to `req.` and `bereq.` variables.

Variables belonging to the backend request (`bereq.`) are assigned with values from the original request (`req.`). However, their values may slightly differ, because Varnish may modify HTTP requests methods. For example, client requests with `HEAD` methods may be converted to backend requests with `GET` methods.



Many but not all of the variables are self-explanatory. To get more information about a particular variable, consult the VCL man page or ask the instructor at the course.



7.8 Summary of VCL

- VCL provides a state machine for controlling Varnish
- Each request is handled independently
- Building a VCL file is done one line at a time

VCL is all about policy. By providing a state machine which you can hook into, VCL allows you to affect the handling of any single request almost anywhere in the execution chain.

This provides pros and cons as any other programming language. This book is not a complete reference guide to how you can deal with every possible scenario in VCL, but on the other hand, if you master the basics of VCL you can solve complex problems that nobody has thought about before. And you can usually do it without requiring too many different sources of documentation.

Whenever you are working on VCL, you should think of what that exact line you are writing has to do. The best VCL is built by having many independent sections that do not interfere with each other more than what they have to.

Remember that there is a built-in VCL. **If your own VCL code does not reach a return statement, the built-in VCL subroutine is executed after yours.** If you just need a little modification of a subroutine, you can use the code from `builtin.vcl` as a template.



8 VCL Built-in Subroutines

- Cover the VCL built-in subroutines: `vcl_recv`, `vcl_pass`, `vcl_backend_fetch`, `vcl_backend_response`, `vcl_hash`, `vcl_hit`, `vcl_miss`, `vcl_deliver`, and `vcl_synth`
- If your VCL code does not reach a return statement, the built-in VCL subroutine is executed after yours.

This chapter covers the VCL subroutines where you customize the behavior of Varnish. However, this chapter does not define caching policies. VCL subroutines can be used to: add custom headers, change the appearance of the Varnish error message, add HTTP redirect features in Varnish, purge content, and define what parts of a cached object is unique.

After this chapter, you should know what all the VCL subroutines can be used for. You should also be ready to dive into more advanced features of Varnish and VCL.

Note

It is strongly advised to let the default built-in subroutines whenever is possible. The built-in subroutines are designed with safety in mind, which often means that they handle any flaws in your VCL code in a reasonable manner.

Tip

Looking at the code of built-in subroutines can help you to understand how to build your own VCL code. Built-in subroutines are in the file `/usr/share/doc/varnish/examples/builtin.vcl.gz` or `{varnish-source-code}/bin/varnishd/builtin.vcl`. The first location may change depending on your distro.



8.1 VCL – `vcl_recv`

- Normalize client-input
- Pick a backend web server
- Re-write client-data for web applications
- Decide caching policy based on client-input
- Access Control Lists (ACL)
- Security barriers, e.g., against SQL injection attacks
- Fixing mistakes, e.g., `index.html -> index.html`

`vcl_recv` is the first VCL subroutine executed, right after Varnish has parsed the client request into its basic data structure. `vcl_recv` has four main uses:

1. Modifying the client data to reduce cache diversity. E.g., removing any leading "www." in the `Host:` header.
2. Deciding which web server to use.
3. Deciding caching policy based on client data. E.g., not caching POST requests, only caching specific URLs, etc.
4. Executing re-write rules needed for specific web applications.

In `vcl_recv` you can perform the following terminating actions:

pass: It passes over the cache *lookup*, but it executes the rest of the Varnish request flow. *pass* does not store the response from the backend in the cache.

pipe: This action creates a full-duplex pipe that forwards the client request to the backend without looking at the content. Backend replies are forwarded back to the client without caching the content. Since Varnish does no longer try to map the content to a request, any subsequent request sent over the same keep-alive connection will also be piped. Piped requests do not appear in any log.

hash: It looks up the request in cache.

purge: It looks up the request in cache in order to remove it.

synth - Generate a synthetic response from Varnish. This synthetic response is typically a web page with an error message. *synth* may also be used to redirect client requests.

It's also common to use `vcl_recv` to apply some security measures. Varnish is not a replacement for intrusion detection systems, but can still be used to stop some typical attacks early. Simple [Access Control Lists \(ACLs\)](#) can be applied in `vcl_recv` too.

For further discussion about security in VCL, take a look at the Varnish Security Firewall (VSF) application at <https://github.com/comotion/VSF>. The VSF supports Varnish 3 and above. You may also be interested to look at the Security.vcl project at <https://github.com/comotion/security.vcl>. The Security.vcl project, however, supports only Varnish 3.x.



Tip

The built-in `vc1_recv` subroutine may not cache all what you want, but often it's better not to cache some content instead of delivering the wrong content to the wrong user. There are exceptions, of course, but if you can not understand why the default VCL does not let you cache some content, it is almost always worth it to investigate why instead of overriding it.



8.1.1 Built-in: `vcl_recv`

```
sub vcl_recv {
    if (req.method == "PRI") {
        /* We do not support SPDY or HTTP/2.0 */
        return (synth(405));
    }
    if (req.method != "GET" &&
        req.method != "HEAD" &&
        req.method != "PUT" &&
        req.method != "POST" &&
        req.method != "TRACE" &&
        req.method != "OPTIONS" &&
        req.method != "DELETE") {
        /* Non-RFC2616 or CONNECT which is weird. */
        return (pipe);
    }

    if (req.method != "GET" && req.method != "HEAD") {
        /* We only deal with GET and HEAD by default */
        return (pass);
    }
    if (req.http.Authorization || req.http.Cookie) {
        /* Not cacheable by default */
        return (pass);
    }
    return (hash);
}
```

The built-in VCL for `vcl_recv` is designed to ensure a safe caching policy even with no modifications in VCL. It has two main uses:

1. Only handle recognized HTTP methods.
2. Cache GET and HEAD. Policies for no caching data is to be defined in your VCL.

Built-in VCL code is executed right after any user-defined VCL code, and is always present. You can not remove it. However, the default VCL code will not execute if you use one of the terminating actions: `pass`, `pipe`, `hash`, or `synth`. These terminating actions return control from the VRT (VCL Run-Time) to Varnish.

For a well-behaving Varnish server, most of the logic in the default VCL is needed, and care should be taken when `vcl_recv` is terminated. Consider either replicating all the built-in VCL logic in your own VCL code, or let your client requests be handled by the built-in VCL code.



8.1.2 Example: Basic Device Detection

One way of serving different content for mobile devices and desktop browsers is to run some simple parsing on the *User-Agent* header. The following VCL code is an example to create custom headers. These custom headers differentiate mobile devices from desktop computers.

```
sub vcl_recv {
    if (req.http.User-Agent ~ "iPad" ||
        req.http.User-Agent ~ "iPhone" ||
        req.http.User-Agent ~ "Android") {

        set req.http.X-Device = "mobile";
    } else {
        set req.http.X-Device = "desktop";
    }
}
```

You can read more about different types of device detection at <https://www.varnish-cache.org/docs/trunk/users-guide/devicedetection.html>

This simple VCL will create a request header called *X-Device* which will contain either *mobile* or *desktop*. The web server can then use this header to determine what page to serve, and inform Varnish about it through *Vary: X-Device*.

It might be tempting to just send *Vary: User-Agent*, but that requires you to normalize the *User-Agent* header itself because there are many tiny variations in the description of *similar User-Agents*. This normalization, however, leads to loss of detailed information of the browser. If you pass the *User-Agent* header without normalization, the cache size may drastically inflate because Varnish would keep possibly hundreds of different variants per object and per tiny *User-Agent* variants. For more information on the *Vary* HTTP response header, see the [Vary](#) section.

Note

If you do use *Vary: X-Device*, you might want to send *Vary: User-Agent* to the users *after* Varnish has used it. Otherwise, intermediary caches will not know that the page looks different for different devices.



8.1.3 Exercise: Rewrite URLs and Host headers

1. Copy the original *Host*-header (`req.http.Host`) and URL (`req.url`) to two new request headers: `req.http.x-host` and `req.http.x-url`.
2. Ensure that *www.example.com* and *example.com* are cached as one, using `regsub()`.
3. Rewrite all URLs under *http://sport.example.com* to *http://example.com/sport/*. For example: *http://sport.example.com/article1.html* to *http://example.com/sport/article1.html*.
4. Use `varnishlog` to verify the result.

Extra: Make sure `/` and `/index.html` is cached as one object. How can you verify that it is, without changing the content?

Extra 2: Make the redirection work for any domain with *sport.* at the front. E.g: *sport.example.com*, *sport.foobar.example.net*, *sport.blatti*, etc.

For the first point, use `set req.http.headername = "value";` or `set req.http.headername = regsub(...);`.

In point 2, change `req.http.host` by calling the function `regsub(str, regex, sub)`. *str* is the input string, in this case, `req.http.host`. *regex* is the regular-expression matching whatever content you need to change. Use `^` to match what begins with *www*, and `\.` to finish the regular-expression, i.e. `^www..` *sub* is what you desire to change it with, an empty string `"` can be used to remove what matches *regex*.

In point 3, you can check for host headers with a specific domain name: `if (req.http.host == "sport.example.com")`. An alternative is to check for all hosts that start with *sport*, regardless the domain name: `if (req.http.host ~ "^sport\."`). In the first case, setting the host header is straight forward: `set req.http.host = "example.com"`. In the second case, you can set the host header by removing the string that precedes the domain name `set req.http.host = regsub(req.http.host, "^sport\.", "");` Finally, you rewrite the URL in this way: `set req.url = regsub(req.url, "^", "/sport");`

To simulate client requests, you can issue the following command:

```
http -p hH --proxy=http:http://localhost sport.example.com/article1.html
```

To verify your result, you can issue the following command:

```
``varnishlog -i ReqHeader,ReqURL``.
```

Tip

Remember that `man vcl` contains a reference manual with the syntax and details of functions such as `regsub(str, regex, sub)`. We recommend you to leave the default VCL file untouched, and create a new file for your VCL code. Remember to update the location of the VCL file in the Varnish configuration file, and restart Varnish.





8.1.4 Solution: Rewrite URLs and Host headers

```
http -p hH --proxy=http:http://localhost sport.example.com/index.html
```

```
varnishlog -i ReqHeader,ReqURL
```

```
sub vcl_recv {

    set req.http.x-host = req.http.host;
    set req.http.x-url = req.url;

    set req.http.host = regsub(req.http.host, "^www\.","");

    /* Alternative 1 */
    if (req.http.host == "sport.example.com") {
        set req.http.host = "example.com";
        set req.url = "/sport" + req.url;
    }

    /* Alternative 2 */
    if (req.http.host ~ "^sport\.") {
        set req.http.host = regsub(req.http.host,"^sport\.","");
        set req.url = regsub(req.url, "^", "/sport");
    }
}
```



8.2 VCL – `vc1_pass`

- Called upon entering *pass* mode

```
sub vc1_pass {  
    return (fetch);  
}
```

The `vc1_pass` subroutine is called after a previous subroutine returns the *pass* action. This action sets the request in *pass* mode. `vc1_pass` typically serves as an important *catch-all* for features you have implemented in `vc1_hit` and `vc1_miss`.

`vc1_pass` may return three different actions: *fetch*, *synth*, or *restart*. When returning the *fetch* action, the ongoing request proceeds in *pass* mode. Fetched objects from requests in *pass* mode are not cached, but passed to the client. The *synth* and *restart* return actions call their corresponding subroutines.



8.2.1 *hit-for-pass*

- Used when an object should not be cached
- *hit-for-pass* object instead of fetched object
- Has TTL

Some requested objects should not be cached. A typical example is when a requested page contains a `Set-Cookie` response header, and therefore it must be delivered only to the client that requests it. In this case, you can tell Varnish to create a *hit-for-pass* object and stores it in the cache, instead of storing the fetched object. Subsequent requests are processed in *pass* mode.

When an object should not be cached, the `beresp.uncacheable` variable is set to *true*. As a result, the *cache process* keeps a hash reference to the *hit-for-pass* object. In this way, the lookup operation for requests translating to that hash find a *hit-for-pass* object. Such requests are handed over to the `vc1_pass` subroutine, and proceed in *pass* mode.

As any other cached object, *hit-for-pass* objects have a TTL. Once the object's TTL has elapsed, the object is removed from the cache.



8.3 VCL – `vcl_backend_fetch` and `vcl_backend_response`

- Sanitize server-response
- Override cache duration

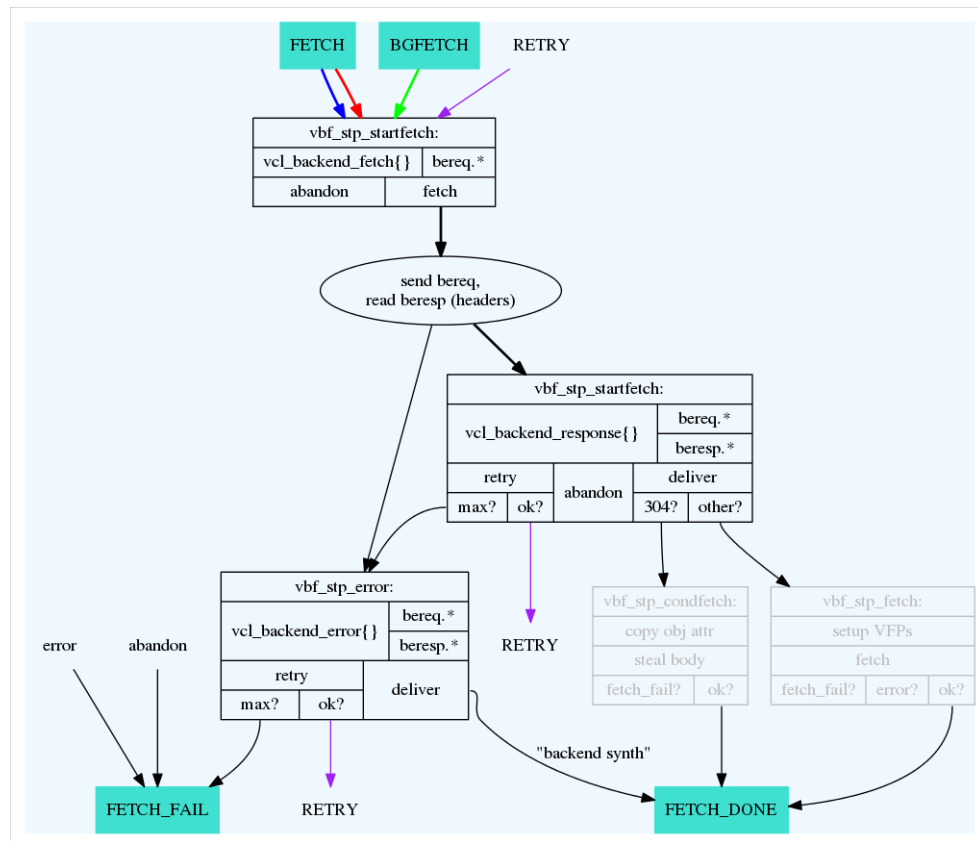


Figure 23: Varnish Request Flow for the Backend Worker Thread.

Figure 23 shows the `vcl_backend_fetch`, `vcl_backend_response` and `vcl_backend_error` subroutines. These subroutines are the backend-counterparts to `vcl_recv`. You can use data provided by the client in `vcl_recv` or even `vcl_backend_fetch` to decide on caching policy. An important difference is that you have access to `bereq.*` variables in `vcl_backend_fetch`.

`vcl_backend_fetch` can be called from `vcl_miss` or `vcl_pass`. When `vcl_backend_fetch` is called from `vcl_miss`, the fetched object may be cached. If `vcl_backend_fetch` is called from `vcl_pass`, the fetched object is **not** cached even if `obj.ttl` or `obj.keep` variables are greater than zero.



A relevant variable is `bereq.uncacheable`. This variable indicates whether the object requested from the backend may be cached or not. However, all objects from *pass* requests are never cached, regardless the `bereq.uncacheable` variable.

`vcl_backend_fetch` has two possible terminating actions, *fetch* or *abandon*. The *fetch* action sends the request to the backend, whereas the *abandon* action calls the `vcl_synth` routine. The built-in `vcl_backend_fetch` subroutine simply returns the *fetch* action. The backend response is processed by `vcl_backend_response` or `vcl_backend_error`.

Figure 24 shows that `vcl_backend_response` may terminate with one of the following actions: *deliver*, *abandon*, or *retry*. The *deliver* terminating action may or may not insert the object into the cache depending on the response of the backend.

Backends might respond with a 304 HTTP headers. 304 responses happen when the requested object has not been modified since the timestamp `If-Modified-Since` in the HTTP header. If the request hits a non fresh object (see Figure 2), Varnish adds the `If-Modified-Since` header with the value of `t_origin` to the request and sends it to the backend.

304 responses do not contain a message-body. Thus, Varnish tries to *steal* the body from cache, merge it with the header response and deliver it. This process updates the attributes of the cached object.

Typical tasks performed in `vcl_backend_fetch` or `vcl_backend_response` include:

- Overriding cache time for certain URLs
- Stripping Set-Cookie headers that are not needed
- Stripping bugged Vary headers
- Adding helper-headers to the object for use in banning (more information in later sections)
- Applying other caching policies



8.3.1 *vcl_backend_response*

built-in *vcl_backend_response*

```
sub vcl_backend_response {
    if (beresp.ttl <= 0s ||
        beresp.http.Set-Cookie ||
        beresp.http.Surrogate-control ~ "no-store" ||
        (!beresp.http.Surrogate-Control &&
            beresp.http.Cache-Control ~ "no-cache|no-store|private") ||
        beresp.http.Vary == "*") {
        /*
         * Mark as "Hit-For-Pass" for the next 2 minutes
         */
        set beresp.ttl = 120s;
        set beresp.uncacheable = true;
    }
    return (deliver);
}
```

The *vcl_backend_response* built-in subroutine is designed to avoid caching in conditions that are most probably undesired. For example, it avoids caching responses with cookies, i.e., responses with *Set-cookie* HTTP header field. This built-in subroutine also avoids *request serialization* described in the [Waiting State](#) section.

To avoid *request serialization*, *beresp.uncacheable* is set to *true*, which in turn creates a *hit-for-pass* object. The [hit-for-pass](#) section explains in detail this object type.

If you still decide to skip the built-in *vcl_backend_response* subroutine by having your own and returning *deliver*, be sure to **never** set *beresp.ttl* to 0. If you skip the built-in subroutine and set 0 as TTL value, you are effectively removing objects from cache that could eventually be used to avoid *request serialization*.

Note

Varnish 3.x has a *hit_for_pass* return action. In Varnish 4, this action is achieved by setting *beresp.uncacheable* to *true*. The [hit-for-pass](#) section explains this in more detail.



8.3.2 The Initial Value of *beresp.ttl*

Before Varnish runs `vcl_backend_response`, the `beresp.ttl` variable has already been set to a value. `beresp.ttl` is initialized with the first value it finds among:

- The `s-maxage` variable in the `Cache-Control` response header field
- The `max-age` variable in the `Cache-Control` response header field
- The `Expires` response header field
- The `default_ttl` parameter.

Only the following status codes will be cached by default:

- 200: OK
- 203: Non-Authoritative Information
- 300: Multiple Choices
- 301: Moved Permanently
- 302: Moved Temporarily
- 304: Not modified
- 307: Temporary Redirect
- 410: Gone
- 404: Not Found

You can cache other status codes than the ones listed above, but you have to set the `beresp.ttl` to a positive value in `vcl_backend_response`. Since `beresp.ttl` is set before `vcl_backend_response` is executed, you can modify the directives of the `Cache-Control` header field without affecting `beresp.ttl`, and vice versa. `Cache-Control` directives are defined in RFC7234 Section 5.2.

A backend response may include the response header field of maximum age for shared caches `s-maxage`. This field overrides all `max-age` values throughout all Varnish servers in a multiple Varnish-server setup. For example, if the backend sends `Cache-Control: max-age=300, s-maxage=3600`, all Varnish installations will cache objects with an `Age` value less or equal to 3600 seconds. This also means that responses with `Age` values between 301 and 3600 seconds are not cached by the clients' web browser, because `Age` is greater than `max-age`.

A sensible approach is to use the `s-maxage` directive to instruct Varnish to cache the response. Then, remove the `s-maxage` directive using `regsub()` in `vcl_backend_response` before delivering the response. In this way, you can safely use `s-maxage` as the cache duration for Varnish servers, and set `max-age` as the cache duration for clients.



Warning

Bear in mind that removing or altering the `Age` response header field may affect how responses are handled downstream. The impact of removing the `Age` field depends on the HTTP implementation of downstream intermediaries or clients.

For example, imagine that you have a three Varnish-server serial setup. If you remove the `Age` field in the first Varnish server, then the second Varnish server will assume `Age=0`. In this case, you might inadvertently be delivering stale objects to your client.



8.3.3 Example: Setting TTL of .jpg URLs to 60 seconds

```
sub vcl_backend_response {  
    if (bereq.url ~ "\.jpg$") {  
        set beresp.ttl = 60s;  
    }  
}
```

The above example caches all URLs ending with .jpg for 60 seconds. Keep in mind that the built-in VCL is still executed. That means that images with a `Set-Cookie` field are not cached.



8.3.4 Example: Cache .jpg for 60 seconds only if s-maxage is not present

```
sub vcl_backend_response {  
    if (beresp.http.cache-control !~ "s-maxage" && bereq.url ~ "\.jpg$") {  
        set beresp.ttl = 60s;  
    }  
}
```

The purpose of the above example is to allow a gradual migration to using a backend-controlled caching policy. If the backend does not supply `s-maxage`, and the URL is a jpg file, then Varnish sets `beresp.ttl` to 60 seconds.

The `Cache-Control` response header field can contain a number of directives. Varnish parses this field and looks for `s-maxage` and `max-age`.

By default, Varnish sets `beresp.ttl` to the value of `s-maxage` if found. If `s-maxage` is not found, Varnish uses the value `max-age`. If neither exists, Varnish uses the `Expires` response header field to set the TTL. If none of those header fields exist, Varnish uses the default TTL, which is 120 seconds.

The default parsing and TTL assignment are done before `vcl_backend_response` is executed. The TTL changing process is recorded in the `TTL` tag of `varnishlog`.



8.3.5 Exercise: Avoid caching a page

Write a VCL which avoids caching the index page at all. It should cover both accessing `/` and `/index.html`

When trying this out, remember that Varnish keeps the *Host*-header in `req.http.host` and the part after the hostname in `req.url`. For `http://www.example.com/index.html`, the `http://` part is not seen by Varnish at all, but `req.http.host` has the value of `www.example.com` and `req.url` the value of `/index.html`. Note how the leading `/` is included in `req.url`.



8.3.6 Solution: Avoid caching a page

```
// Suggested solution A
sub vcl_recv {
    if (req.url ~ "^/index\.html" || req.url ~ "^/$") {
        return(pass);
    }
}
// Suggested solution B
sub vcl_backend_fetch {
    if (breq.url ~ "^/index\.html" || breq.url ~ "^/$") {
        set bereq.uncacheable = true;
        return(fetch);
    }
}
```

Usually it is most convenient to do as much as possible in `vcl_recv`. The usage of `bereq.uncacheable` in `vcl_backend_fetch` creates a *hit-for-pass* object. See the [hit-for-pass](#) section for detailed description about this type of object.



8.3.7 Exercise: Either use *s-maxage* or set *TTL* by file type

Write a VCL that:

- uses `Cache-Control: s-maxage` when present,
- caches `.jpg` for 30 seconds if `s-maxage` is not present,
- caches `.html` for 10 seconds if `s-maxage` isn't present, and
- removes the `Set-Cookie` header field if `s-maxage` OR the above rules indicate that Varnish should cache.

Tip

Divide and conquer! Most somewhat complex VCL tasks are easily solved when you divide the tasks into smaller problems and solve them individually. Try solving each part of the exercise by itself first.

Note

Varnish automatically parses `s-maxage` for you, so you only need to check if it is there or not. Remember that if `s-maxage` is present, Varnish has already used it to set `beresp.ttl`.



8.3.8 Solution: Either use *s-maxage* or set *ttl* by file type

```
sub vcl_backend_response {
    if (beresp.http.cache-control !~ "s-maxage") {
        if (bereq.url ~ "\.jpg(\?|$)") {
            set beresp.ttl = 30s;
            unset beresp.http.Set-Cookie;
        }
        if (bereq.url ~ "\.html(\?|$)") {
            set beresp.ttl = 10s;
            unset beresp.http.Set-Cookie;
        }
    } else {
        if (beresp.ttl > 0s) {
            unset beresp.http.Set-Cookie;
        }
    }
}
```

There are many ways to solve this exercise, and this solution is only one of them. The first part checks that *s-maxage* is *not* present, then handles .jpg and .html files - including cookie removal. The second part checks if *s-maxage* caused Varnish to set a positive TTL and consider it cacheable. Then, we remove the *Set-Cookie* header field.



8.4 VCL – `vcl_hash`

- Defines what is unique about a request.
- Executed after `vcl_recv` returns a `hash` action keyword.

```
sub vcl_hash {  
    hash_data(req.url);  
    if (req.http.host) {  
        hash_data(req.http.host);  
    } else {  
        hash_data(server.ip);  
    }  
    return (lookup);  
}
```

`vcl_hash` defines the hash key to be used for a cached object. Hash keys differentiate one cached object from another. The default VCL for `vcl_hash` adds the hostname or IP address, and the requested URL to the cache hash.

One usage of `vcl_hash` is to add a user-name in the cache hash to identify user-specific data. However, be warned that caching user-data should only be done cautiously. A better alternative might be to hash cache objects per session instead.

The `vcl_hash` subroutine returns the `lookup` action keyword. Unlike other action keywords, `lookup` is an operation, not a subroutine. Depending on what `lookup` finds in the cache, it has four possible outcomes: `hit`, `miss`, `hit-for-pass`, or `purge`.

When the `lookup` operation does not match any hash, it creates an object with a *busy* flag and inserts it in cache. Then, the request is sent to the `vcl_miss` subroutine. The *busy* flag is removed once the request is handled, and the object is updated with the response from the backend.

Subsequent similar requests that hit *busy* flagged objects are sent into a *waiting list*. This waiting list is designed to improve response performance, and it is explained in the [Waiting State](#) section.

Note

One cache hash may refer to one or many object variations. Object variations are created based on the [Vary](#) header field. It is a good practice to keep several variations under one cache hash, than creating one hash per variation.



8.5 VCL – `vcl_hit`

- Executed after the lookup operation, called by `vcl_hash`, finds (hits) an object in the cache.

```
sub vcl_hit {
    if (obj.ttl >= 0s) {
        // A pure unadultered hit, deliver it
        return (deliver);
    }
    if (obj.ttl + obj.grace > 0s) {
        // Object is in grace, deliver it
        // Automatically triggers a background fetch
        return (deliver);
    }
    // fetch & deliver once we get the result
    return (fetch);
}
```

The `vcl_hit` subroutine typically terminate by calling `return()` with one of the following keywords: `deliver`, `restart`, or `synth`.

`deliver` returns control to `vcl_deliver` if the TTL + grace time of an object has not elapsed. If the elapsed time is more than the TTL, but less than the TTL + grace time, then `deliver` calls for *background fetch* in parallel to `vcl_deliver`. The background fetch is an asynchronous call that inserts a *fresher* requested object in the cache. Grace time is explained in the [Grace Mode](#) section.

`restart` starts again the transaction, and increases the restart counter. If the number of restarts is higher than `max_restarts` counter, Varnish emits a *guru meditation* error.

`synth(status code, reason)` returns the specified status code to the client and abandon the request.



8.6 VCL – `vcl_miss`

- Subroutine called if a requested object is not found by the lookup operation.
- Contains policies to decide whether or not to attempt to retrieve the document from the backend, and which backend to use.

```
sub vcl_miss {  
    return (fetch);  
}
```

The subroutines `vcl_hit` and `vcl_miss` are closely related. It is rare that you customize them, because modification of HTTP request headers is typically done in `vcl_recv`. However, if you do not wish to send a *X-Varnish* header to the backend server, you can remove it in `vcl_miss` or `vcl_pass`. For that case, you can use `unset bereq.http.x-varnish;`.



8.7 VCL – `vcl_deliver`

- Common last exit point for all request workflows, except requests through `vcl_pipe`
- Often used to add and remove debug-headers

```
sub vcl_deliver {  
    return (deliver);  
}
```

The `vcl_deliver` subroutine is simple, and it is also very useful to modify the output of Varnish. If you need to remove a header, or add one that is not supposed to be stored in the cache, `vcl_deliver` is the place to do it.

The variables most useful and common to modify in `vcl_deliver` are:

`resp.http.*`

Headers that are sent to the client. They can be set and unset.

`resp.status`

The status code (200, 404, 503, etc).

`resp.reason`

The HTTP status message that is returned to the client.

`obj.hits`

The count of cache-hits on this object. Therefore, a value of 0 indicates a miss. This variable can be evaluated to easily reveal whether a response comes from a cache hit or miss.

`req.restarts`

The number of restarts issued in VCL - 0 if none were made.



8.8 VCL – `vcl_synth`

- Used to generate content from within Varnish, without talking to the backend
- Error messages go here by default
- Other use cases: Redirecting users (301/302 redirects)

```
sub vcl_synth {
    set resp.http.Content-Type = "text/html; charset=utf-8";
    set resp.http.Retry-After = "5";
    synthetic( {"<!DOCTYPE html>
<html>
  <head>
    <title>"} + resp.status + " " + resp.reason + {"</title>
</head>
  <body>
    <h1>Error "} + resp.status + " " + resp.reason + {"</h1>
    <p>"} + resp.reason + {"</p>
    <h3>Guru Meditation:</h3>
    <p>XID: "} + req.xid + {"</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
"} );
    return (deliver);
}
```

To make a distinction between VCL synthetic responses and internally generated errors (when trying to fetch an object), there are two subroutines that handle errors in Varnish. One is `vcl_synth`, and the other is `vcl_backend_error`. To return control to `vcl_synth`, call the `synth()` function like this:

```
return (synth(status_code, "reason"));
```

Note that the syntax `synth` is not a keyword, but a function with arguments.

You must explicitly return the `status_code` and `reason` arguments for `vcl_synth`. Setting headers on synthetic response bodies are done on `resp.http`.

Note

Note how you can use `{"` and `"}` to make multi-line strings. This is not limited to the `synthetic()` function, but can be used anywhere.



Note

A `vcl_synth` defined object is never stored in cache, contrary to a `vcl_backend_error` defined object, which may end up in cache. `vcl_synth` and `vcl_backend_error` replace `vcl_error` from Varnish 3.



8.8.1 Example: Redirecting requests with `vcl_synth`

```
sub vcl_recv {
    if (req.http.host == "www.example.com") {
        set req.http.Location = "http://example.com" + req.url;
        return (synth(750, "Permanently moved"));
    }
}

sub vcl_synth {
    if (resp.status == 750) {
        set resp.http.location = req.http.Location;
        set resp.status = 301;
        return (deliver);
    }
}
```

Redirecting with VCL is fairly easy – and fast. Basic HTTP redirects work when the HTTP response is either *301 Moved Permanently* or *302 Found*. These response have a Location header field telling the web browser where to redirect.

Note

The *301* response can affect how browsers prioritize history and how search engines treat the content. *302* responses are temporary and do not affect search engines as *301* responses do.



8.9 Exercise: Modify the HTTP response header fields

- Add a header stating either HIT or MISS
- "Rename" the Age header to X-Age.



8.9.1 Solution: Modify the HTTP response header fields

```
sub vcl_deliver {
    set resp.http.X-Age = resp.http.Age;
    unset resp.http.Age;

    if (obj.hits > 0) {
        set resp.http.X-Cache = "HIT";
    } else {
        set resp.http.X-Cache = "MISS";
    }
}
```

It is safer to make sure a variable has a sensible value before using it to make a string. Therefore, we check that `obj.hits > 0` (and not just `obj.hits != 0`) before using it.

There have been bugs in string-conversion. Those bugs happened when the variable to be converted to string had an unexpected value. This applies to all variables – and all languages for that matter.



8.10 Exercise: Change the error message

- Make the default error message more friendly.



8.10.1 Solution: Change the error message

```
/* Change your backend configuration to provoke a 503 error */
backend default {
    .host = "127.0.0.1";
    .port = "8081";
}

/* Customize error responses */
sub vcl_backend_error {
    if (beresp.status == 503){
        set beresp.status = 200;
        synthetic( {"
            <html><body><!-- Here goes a more friendly error message. --></body></html>
        " } );
        return (deliver);
    }
}
```

The suggested solution forces a 503 error by misconfiguring `.port` in the *default* backend.



9 Cache Invalidation

- Cache invalidation is an important part of your cache policy
- Varnish will automatically invalidate expired objects
- You can however pro-actively invalidate objects with Varnish
- You should define those rules **before caching objects** in production environments

There are four mechanisms to invalidate caches in Varnish:

1. HTTP PURGE

- Use the `vcl_purge` subroutine
- Invalidate caches explicitly using objects' hashes
- `vcl_purge` is called via `return(purge)` from `vcl_recv`
- `vcl_purge` removes all variants of an object from cache, freeing up memory
- The `restart` return action can be used to update immediately a purged object

2. Banning

- Use the built-in function `ban(expression)`
- Invalidates objects in cache that match the regular-expression
- Does not necessarily free up memory at once
- Also accessible from the management interface

3. Force Cache Misses

- Use `req.hash_always_miss` in `vcl_recv`
- If set to true, Varnish disregards any existing objects and always (re)fetches from the backend
- May create multiple objects as side effect
- Does not necessarily free up memory at once

4. Hashtwo -> Varnish Plus only!

- For websites with the need for cache invalidation at a very large scale
- Varnish Software's implementation of surrogate keys
- Flexible cache invalidation based on cache tags



Which to use when?

Whenever you deal with a cache, you have to eventually deal with the challenge of cache invalidation, or refreshing content. There are many motives behind such a task. Varnish addresses the problem in several slightly different ways.

To decide which cache invalidation mechanism to use, answer the following questions:

- Am I invalidating one specific object, or many?
- Do I need to free up memory, or just replace the content?
- How long time does it take to replace the content?
- Is this a regular task, or a one-off task?

The rest of the chapter gives you the information to pickup the most suitable mechanisms.



9.1 HTTP PURGE

- If you know exactly what to remove, use `HTTP PURGE`
- Frees up memory, removes all `Vary:-`variants of the object
- Leaves it to the next client to refresh the content
- Often combined with `return(restart);`

A purge is what happens when you pick out an object from the cache and discard it along with its variants. A resource can exist in multiple `Vary:-`variants. For example, you could have a desktop version, a tablet version and a smartphone version of your site, and use the `Vary` HTTP header field in combination with device detection to store different variants of the same resource.

Usually a purge is invoked through HTTP with the method `PURGE`. A `HTTP PURGE` is another request method just as `HTTP GET`. Actually, you can call the `PURGE` method whatever you like, but `PURGE` has become the de-facto naming standard. Squid, for example, uses the `PURGE` method name for the same purpose.

Purges apply to a specific object, since they use the same lookup operation as in `vc1_hash`. Therefore, purges cannot use regular-expressions. A positive effect is that purges find and remove objects really fast!

The down-side of using `PURGE` is that you evict content from cache before you know if Varnish can fetch a new copy from the backend. That means that if you purge some objects and the backend is down, Varnish will end up having no copy of the content.



9.1.1 VCL – *vcl_purge*

- You may add actions to be executed once the object and its variants is purged
- Called after the purge has been executed

```
sub vcl_purge {  
    return (synth(200, "Purged"));  
}
```

Note

Cache invalidation with purges is done via `return (purge);` from `vcl_recv` in Varnish 4. The `purge;` keyword from Varnish 3 has been retired.



9.1.2 Example: PURGE

In order to support purging in Varnish, you need the following VCL in place.

```
acl purgers {
    "127.0.0.1";
    "192.168.0.0"/24;
}

sub vcl_recv {
    # allow PURGE from localhost and 192.168.0...

    if (req.method == "PURGE") {
        if (!client.ip ~ purgers) {
            return (synth(405));
        }
        return (purge);
    }
}
```

Test your VCL by issuing::

```
http -p hH --proxy=http://localhost PURGE www.example.com
```

`acl` is a reserved keyword that is used to create [Access Control Lists \(ACLs\)](#). ACLs are used to control which client IP addresses are allowed to purge cached objects.

Note the `purge` return action in `vcl_recv`. This action ends execution of `vcl_recv` and jumps to `vcl_hash`. When `vcl_hash` calls `return(lookup)`, Varnish purges the object and then calls `vcl_purge`.



9.1.3 Exercise: *PURGE* an article from the backend

- Send a `PURGE` request to Varnish from your backend server after an article is published.
- Simulate the article publication.
- The result is that the article is evicted in Varnish.

Now you know that purging can be as easy as sending a specific HTTP request. You are provided with `article.php`, which fakes an article. It is recommended to create a separate php file to implement purging.

article.php

```
<?php
header("Cache-Control: public, must-revalidate, max-age=3600, s-maxage=3600");
$date = new DateTime();
$now = $date->format( DateTime::RFC2822 );
?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h1>This is an article, cached for 1 hour</h1>

    <h2>Now is <?php echo $now; ?></h2>
    <a href="<?=$_SERVER['PHP_SELF']?>">Refresh this page</a>
  </body>
</html>
```

Tip

Remember to place your php files under `/var/www/html/`.



9.1.4 Solution: *PURGE* an article from the backend

purgearticle.php

```
<?php
header( 'Content-Type: text/plain' );
header( 'Cache-Control: max-age=0' );
$hostname = 'localhost';
$port     = 80;
$URL      = '/article.php';
$debug    = true;

print "Updating the article in the database ...\n";
purgeURL( $hostname, $port, $URL, $debug );

function purgeURL( $hostname, $port, $purgeURL, $debug )
{
    $finalURL = sprintf(
        "http://%s:%d%s", $hostname, $port, $purgeURL
    );

    print( "Purging ${finalURL}\n" );

    $curlOptionList = array(
        CURLOPT_RETURNTRANSFER => true,
        CURLOPT_CUSTOMREQUEST  => 'PURGE',
        CURLOPT_HEADER          => true ,
        CURLOPT_NOBODY          => true,
        CURLOPT_URL              => $finalURL,
        CURLOPT_CONNECTTIMEOUT_MS => 2000
    );

    $fd = false;
    if( $debug == true ) {
        print "\n---- Curl debug ----\n";
        $fd = fopen("php://output", 'w+');
        $curlOptionList[CURLOPT_VERBOSE] = true;
        $curlOptionList[CURLOPT_STDERR]  = $fd;
    }

    $curlHandler = curl_init();
    curl_setopt_array( $curlHandler, $curlOptionList );
    curl_exec( $curlHandler );
    curl_close( $curlHandler );
    if( $fd !== false ) {
        fclose( $fd );
    }
}
```



```
}  
?>
```



solution-purge-from-backend.vcl

```
acl purgers {  
    "127.0.0.1";  
}  
  
sub vcl_recv {  
    if (req.method == "PURGE") {  
        if (!client.ip ~ purgers) {  
            return (synth(405, "Not allowed."));  
        }  
        return (purge);  
    }  
}
```



9.2 PURGE with restart return action

- Start the VCL processing again from the top of `vcl_recv`
- Any changes made are kept

```
acl purgers {
    "127.0.0.1";
    "192.168.0.0"/24;
}

sub vcl_recv {
    # allow PURGE from localhost and 192.168.0...
    if (req.restarts == 0) {
        unset req.http.X-Purger;
    }

    if (req.method == "PURGE") {
        if (!client.ip ~ purgers) {
            return (synth(405, "Purging not allowed for " + client.ip));
        }
        return (purge);
    }
}

sub vcl_purge {
    set req.method = "GET";
    set req.http.X-Purger = "Purged";
    return (restart);
}

sub vcl_deliver {
    if (req.http.X-Purger) {
        set resp.http.X-Purger = req.http.X-Purger;
    }
}
```

The `restart` return action allows Varnish to re-run the VCL state machine with different variables. This is useful in combination with `PURGE`, in the way that a purged object can be immediately restored with a new fetched object.

Every time a *restart* occurs, Varnish increments the `req.restarts` counter. If the number of restarts is higher than the `max_restarts` parameter, Varnish emits a *guru meditation* error. In this way, Varnish safe guards against infinite loops.



Warning

Restarts are likely to cause a hit against the backend, so do not increase `max_restarts` thoughtlessly.



9.3 Banning

- Use `ban` to prevent Varnish from serving a cached object
- Does not free up memory
- Examples in the `varnishadm` command line interface:

```
• ban req.url ~ /foo
• ban req.http.host ~ example.com && obj.http.content-type ~ text
• ban.list
```

- Example in VCL:

```
• ban("req.url ~ /foo");
```

- Example of VCL code to catch the HTTP `BAN` request method:

```
sub vcl_recv {
    if (req.method == "BAN") {
        ban("req.http.host == " + req.http.host +
            " && req.url == " + req.url);
        # Throw a synthetic page so the request won't go to the backend.
        return(synth(200, "Ban added"));
    }
}
```

Banning in the context of Varnish refers to adding a *ban expression* that prohibits Varnish to serve certain objects from the cache. Ban expressions are more useful when using regular-expressions.

Bans work on objects already in the cache, i.e., it does not prevent new content from entering the cache or being served. Cached objects that match a ban are marked as obsolete. Obsolete objects are expunged by the expiry thread like any other object with `obj.ttl == 0`.

Ban expressions match against `req.*` or `obj.*` variables. Think about a ban expression as; "the requested URL starts with /sport", or "the cached object has a Server-header matching lighttpd". You can add ban expressions in VCL syntax via VCL code, HTTP request method, or CLI.

Ban expressions are inserted into a ban-list. The ban-list contains:

- ID of the ban,
- timestamp when the ban entered the ban list,
- counter of objects that have matched the ban expression,
- a `c` flag for *completed* that indicates whether a ban is invalid because it is duplicated,
- the ban expression.

To inspect the current ban-list, issue the `ban.list` command in the CLI:



```
0xb75096d0 1318329475.377475    10    obj.http.x-url ~ test0
0xb7509610 1318329470.785875    20C   obj.http.x-url ~ test1
```

Varnish tests bans whenever a request hits a cached object. A cached object is checked only against bans added after the last checked ban. That means that each object checks against a ban expression only once.

Bans that match only against `obj.*` are also checked by a background worker thread called the *ban lurker*. The parameter `ban_lurker_sleep` controls how often the *ban lurker* tests `obj.*` bans. The ban lurker can be disabled by setting `ban_lurker_sleep` to 0.

Note

You may accumulate a lot of ban expressions based in `req.*` variables if you have many objects with long TTL that are seldom accessed. This accumulation occurs because bans are kept until all cached objects have been checked against them. This might impact CPU usage and thereby performance.

Therefore, we recommend you to avoid `req.*` variables in your ban expressions, and to use `obj.*` variables instead. Ban expressions using only `obj.*` are called *lurker-friendly bans*.



Note

If the cache is completely empty, only the last added ban stays in the ban-list.

Tip

You can also execute ban expressions via the [Varnish Administration Console \(VAC\)](#).



 OVERVIEW CONFIGURE BANS USERS SUPPORT MESSAGES Signed as [vac](#)  Sign out

BAN EXPRESSION:

req.url ~ /

GROUP:

Test

EXECUTE BAN

Recent Bans

Expression	Group	Date & Time
req.http.host ~ ".*\example.com" && req.url ~ "^/articles" && req.http.content-type == "text/html"	Test	27/04/2015 15:37:09
req.http.host ~ ".*\example.com" && req.url ~ "\.png\$"	Test	27/04/2015 15:37:05
req.url ~ "/articles/"	Test	27/04/2015 15:37:01
req.url ~ /	Test	27/04/2015 15:36:45
req.url ~ /	Test	27/04/2015 15:36:41

LOAD MORE BANS

Bans

Bans is how you invalidate content currently in the cache.

In this view you can add bans that are executed on all the caches in a cache group. The ban text is the complete expression as it would be handed to `ban()` in VCL.

Click on one of the examples below to use:

Ban Expression	Description
<div>req.url ~ /</div>	Empty the whole cache

Figure 24: Executing ban expressions via the *Varnish Administration Console (VAC)*.



9.3.1 Lurker-Friendly Bans

- Ban expressions that match only against `obj.*`
- Evaluated asynchronously by the *ban lurker* thread
- Similar to the concept of *garbage collection*

Ban expressions are checked in two cases: 1) when a request hits a cached object, or 2) when the ban lurker *wakes up*. The first case is efficient only if you know that the cached objects to be banned are frequently accessed. Otherwise, you might accumulate a lot of ban expressions in the ban-list that are never checked. The second case is a better alternative because the ban lurker can help you keep the ban list at a manageable size. Therefore, we recommend you to create ban expressions that are checked by the ban lurker. Such ban expressions are called *lurker-friendly bans*.

Lurker-friendly ban expressions are those that use only `obj.*`, but not `req.*` variables. Since *lurker-friendly ban* expressions lack of `req.*`, you might need to copy some of the `req.*` contents into the `obj` structure. In fact, this copy operation is a mechanism to preserve the context of client request in the cached object. For example, you may want to copy useful parts of the client context such as the requested URL from `req` to `obj`.



The following snippet shows an example on how to preserve the context of a client request in the cached object:

```
sub vcl_backend_response {
    set beresp.http.x-url = bereq.url;
}

sub vcl_deliver {
    # The X-Url header is for internal use only
    unset resp.http.x-url;
}
```

Now imagine that you just changed the blog post template. To invalidate all blog posts, you can then issue a ban such as:

```
$ varnishadm ban 'obj.http.x-url ~ ^/blog'
```

Since it uses a *lurker-friendly ban* expression, the ban inserted in the ban-list will be gradually evaluated against all cached objects until all blog posts are invalidated. The snippet below shows how to insert the same expression into the ban-list in the `vcl_recv` subroutine:

```
sub vcl_recv {
    if (req.method == "BAN") {
        # Assumes the ``X-Ban`` header is a regex,
        # this might be a bit too simple.

        ban("obj.http.x-url ~ " + req.http.x-ban);
        return(synth(200, "Ban added"));
    }
}
```



9.4 Exercise: Write a VCL program using *purge* and *ban*

Write a VCL program that implements both cache invalidation mechanisms: *purge* and *ban*. The *ban* method should use the request headers `req.http.x-ban-url` and `req.http.x-ban-host`, and it should implement *lurker-friendly bans*.

To build further on this, you can also have a `REFRESH` HTTP method that fetches new content, using `req.hash_always_miss`.

To test this exercise, you can use *httpie*. Example commands:

```
http -p hH PURGE http://localhost/testpage
http -p hH BAN http://localhost/ 'X-Ban-Url: .*html$' \
                                'X-Ban-Host: .*\.example\.com'
http -p hH REFRESH http://localhost/testpage
```



9.4.1 Solution: Write a VCL program using purge and ban

```
sub vcl_recv {
    if (req.method == "PURGE") {
        return (purge);
    }

    if (req.method == "BAN") {
        ban("obj.http.x-url ~ " + req.http.x-ban-url +
            " && obj.http.x-host ~ " + req.http.x-ban-host);
        return (synth(200, "Ban added"));
    }

    if (req.method == "REFRESH") {
        set req.method = "GET";
        set req.hash_always_miss = true;
    }
}

sub vcl_backend_response {
    set beresp.http.x-url = bereq.url;
    set beresp.http.x-host = bereq.http.host;
}

sub vcl_deliver {
    unset resp.http.x-url; # Optional, for internal
    unset resp.http.x-host; # use only...
}
```



9.5 Force Cache Misses

- `set req.hash_always_miss = true;` in `vcl_recv`
- Causes Varnish to look the object up in cache, but ignore any copy it finds
- Useful way to do a controlled refresh of a specific object
- If the server is down, the cached object is left untouched
- Depending on Varnish-version, it may leave extra copies in the cache
- Useful to refresh slowly generated content

Setting a request in *pass* mode instructs Varnish to always ask a backend for content, without storing the fetched object into cache. The `vcl_purge` removes old content, but what if the web server is down?

Setting `req.hash_always_miss` to `true` tells Varnish to look up the content in cache, but always miss a hit. This means that Varnish first calls `vcl_miss`, then (presumably) fetches the content from the backend, cache the updated object, and deliver the updated content.

The distinctive behavior of `req.hash_always_miss` occurs when the backend server is down or unresponsive. In this case, the current cached object is untouched. Therefore, client requests that do not enable `req.hash_always_miss` keep getting the old and untouched cached content.

Two important use-cases for using `req.hash_always_miss` are: 1) control who takes the penalty for waiting around for the updated content (e.g. a script you control), and 2) ensure that content is not evicted before there it is updated.

Note

Forcing cache misses do not evict old content. This means that causes Varnish to have multiple copies of the content in cache. The newest copy is always used. If you cache your content for a long period of time, the memory usage increases gradually.



9.6 Hashtwo (Varnish Software Implementation of Surrogate Keys)

- Hashtwo is Varnish Software's implementation of surrogate keys
- Cache invalidation based on cache tags
- Adds patterns easily to be matched against
- Highly scalable
- Hashtwo is distributed as a VMOD for Varnish Plus only!
- Install the Varnish Plus VMODs

The idea is that you can use any arbitrary string for cache invalidation. You can then key your cached objects on, for example, product ID or article ID. In this way, when you update the price of a certain product or a specific article, you have a key to evict all those objects from the cache.

So far, we have discussed *purges* and *bans* as mechanisms for cache invalidation. Two important distinctions between them is that *purges* remove a single object (with its variants), whereas *bans* perform cache invalidation based on matching expressions. However, there are cases where none of these mechanisms are optimal.

Hashtwo creates a second hash key to link cached objects based on cache tags. This hash keys provide the means to invalidate cached objects with common cache tags.

In practice, *Hashtwo* create cache invalidation patterns, which can be tested and invalidated immediately just as *purges* do. In addition, *Hashtwo* is much more efficient than *bans* because of two reasons: 1) looking up *hash keys* is much more efficient than traversing ban-lists, and 2) every time you test a ban expression, it checks every object in the cache that is older than the ban itself.

The hashtwo VMOD is pre-built for supported versions and can be installed using regular package managers from the Varnish Software repositories. Once the repository is in place you can issue the following commands to install the VMOD:

On Debian or Ubuntu:

```
apt-get install libvmod-hashtwo
```

On Red Hat Enterprise Linux:

```
yum install libvmod-hashtwo
```

Finally, you can use this VMOD by importing it in your VCL code:

```
import hashtwo;
```



9.6.1 VCL example using Hashtwo

On an e-commerce site the backend application issues a `X-HashTwo` HTTP header field for every product that is referenced on that page. The header for a certain page might look like this:

```
HTTP/1.1 200 OK
Server: Apache/2.2.15
X-HashTwo: 8155054
X-HashTwo: 166412
X-HashTwo: 234323
```

The VCL example code:

```
import hashtwo;

# In this example the key to be purged is specified on the
# X-HashTwo-Purge header.
sub vcl_recv {
    if (req.http.X-HashTwo-Purge) {
        if (hashtwo.purge(req.http.X-HashTwo-Purge) != 0) {
            return (purge);
        } else {
            return (synth(404, "Key not found"));
        }
    }
}

# Normally the backend is responsible for setting the header.
# If you were to do it in VCL it will look something like this:
sub vcl_backend_response {
    set beresp.http.X-HashTwo = "secondary_hash_key";
}
```

In order to keep the web pages in sync with the database, a trigger is set up in the database. When a stock keeping unit (SKU) is updated, an HTTP request towards the Varnish server is triggered. This request invalidates every cached object with the matching `X-HashTwo` header:

```
GET / HTTP/1.1
Host: www.example.com
X-HashTwo-Purge: 166412
```

Note the `X-HashTwo-Purge` HTTP header field.

Based on the VCL code above, Varnish finds the objects and purge them. After that, Varnish responds with:



```
HTTP/1.1 200 Purged
Date: Thu, 24 Apr 2014 17:08:28 GMT
X-Varnish: 1990228115
Via: 1.1 Varnish
```

The objects are now cleared.

Warning

You should protect purges with ACLs from unauthorized hosts.



9.7 Purge vs. Bans vs. Hashtwo vs. Cache Misses

Table 18: Bans vs. Purge vs. Hashtwo vs. Force Cache Misses

	Bans	Purge	Hashtwo – Surrogate keys	Force Cache Misses
Targets	Objects matching patterns	One specific object (with all its variants)	All objects with a common hashtwo key	One specific object (with all its variants)
Frees memory	Not designed to free memory, but it does free memory after a request hits an object in cache that matches a ban expression, or the ban lruer invalidates cached object.	Immediately	Immediately	No
Scalability	High when used properly, e.g., when using lurker friendly bans, having few cached objects, having a short ban-list, or writting and triggering manually a set of bans that target very well identified objects.	High	High	No. Memory usage increases because old objects are not invalidated.
Flexibility	High	Low	High	Low
CLI	Yes	No	No	No
VCL	Yes	Yes	Yes	Yes
Availability	Varnish Cache	Varnish Cache	Varnish Plus only	Varnish Cache

There is rarely a need to pick only one solution, as you can implement many of them. Some guidelines for selection, though:

- Any frequent automated or semi-automated cache invalidation most likely require VCL changes for the best effect.
- If you need to invalidate more than one item at a time, consider to use *bans* or *hashtwo*.
- If it takes a long time to pull content from the backend into Varnish, consider to use `req.hash_always_miss`.



Note

Purge and Hashtwo work very similar. The main difference is that they have they act on different hash keys.



10 Saving a Request

This chapter is for the system administration course only

Table 19: Connotation of Saving a Request

	Rescue	Economization	Protection
Directors	x	x	
Health Checks	x		
Grace Mode	x	x	
Retry a Request	x		
Tune Backend Properties			x
Access Control Lists (ACL)			x

Varnish offers many mechanisms to save a request. By saving a request we mean:

1. Rescue: mechanisms to handle requests when backends are in problematic situations.
2. Economization: mechanisms to spend less resources, i.e., send less requests to the backend.
3. Protection: mechanisms to restrict access cache invalidation from unauthorized entities.

Table 19 shows how different mechanisms are mapped to their saving meaning. This chapter explains how to make your Varnish setup more robust by using these mechanisms.



10.1 Directors

- Loadable VMOD
- Contains 1 or more backends
- All backends must be known
- Selection methods:
 - round-robin
 - fallback
 - random
 - seeded with a random number
 - seeded with a hash key

Round-robin director example:

```
vcl 4.0;

import directors;    // load the directors VMOD

backend one {
    .host = "localhost";
    .port = "80";
}

backend two {
    .host = "127.0.0.1";
    .port = "81";
}

sub vcl_init {
    new round_robin_director = directors.round_robin();
    round_robin_director.add_backend(one);
    round_robin_director.add_backend(two);

    new random_director = directors.random();
    random_director.add_backend(one, 10); # 2/3 to backend one
    random_director.add_backend(two, 5);  # 1/3 to backend two
}

sub vcl_recv {
    set req.backend_hint = round_robin_director.backend();
}
```



Varnish can have several backends defined, and it can set them together into clusters for load balancing purposes. Backend directors, usually just called directors, provide logical groupings of similar web servers by re-using previously defined backends. A director must have a name.

There are several different director selection methods available, they are: random, round-robin, fallback, and hash. The next backend to be selected depends on the selection method. The simplest directors available are the *round-robin* and the *random* director.

A *round-robin* director takes only a backend list as argument. This director type picks the first backend for the first request, then the second backend for the second request, and so on. Once the last backend have been selected, backends are selected again from the top. If a health probe has marked a backend as sick, a round-robin director skips it.

A *fallback* director will always pick the first backend unless it is sick, in which case it would pick the next backend and so on. A director is also considered a backend so you can actually stack directors. You could for instance have directors for active and passive clusters, and put those directors behind a fallback director.

Random directors are seeded with either a random number or a hash key. Next section explains their commonalities and differences.

Note

Health probes are explain in the [Health Checks](#) section.

Note

Directors are defined as loadable VMODs in Varnish 4. Please see the `vmod_directors` man page for more information.



10.1.1 Random Directors

- *Random* director: seeded with a random number
- *Hash* director: seeded with hash key from typically a URL or a client identity string

Hash director that uses client identity for backend selection

```
sub vcl_init {
    new h = directors.hash();
    h.add_backend(one, 1);    // backend 'one' with weight '1'
    h.add_backend(two, 1);   // backend 'two' with weight '1'
}

sub vcl_recv {
    // pick a backend based on the cookie header of the client
    set req.backend_hint = h.backend(req.http.cookie);
}
```

The *random* director picks a backend randomly. It has one per-backend parameter called *weight*, which provides a mechanism for balancing the selection of the backends. The selection mechanism of the random director may be regarded as traffic distribution if the amount of traffic is the same per request and per backend. The random director also has a director-wide counter called *retries*, which increases every time the director selects a sick backend.

Both, the *random* and *hash* director select a backend randomly. The difference between these two is the seed they use. The *random* director is seeded with a random number, whereas the *hash* director is seeded with a hash key.

Hash directors typically use the requested URL or the client identity (e.g. session cookie) to compute the hash key. Since the hash key is always the same for a given input, the output of the *hash* director is always the same for a given hash key. Therefore, *hash* directors select always the same backend for a given input.

Hash directors are useful to load balance in front of other Varnish caches or other web accelerators. In this way, cached objects are not duplicated across different cache servers.

Note

In Varnish 3 there is a *client* director type, which is removed in Varnish 4. This *client* director type is a special case of the *hash* director. Therefore, the semantics of a *client* director type are achieved using `hash.backend(client.identity)`.



10.2 Health Checks

- Poke your web server every N seconds
- Affects backend selection
- `std.healthy(req.backend_hint)`
- Varnish allows at most *threshold* amount of failed probes within a set of the last *window* probes
- *threshold* and *window* are parameters
- Set using `.probe`
- `varnishlog`: `Backend_health`

```
backend server1 {
    .host = "server1.example.com";
    .probe = {
        .url = "/";
        .timeout = 1s;
        .interval = 4s;
        .window = 5;
        .threshold = 3;
    }
}
```

You can define a health check for each backend. A health check defines a *probe* to verify whether a backend replies on a given URL every given interval.

The above example causes Varnish to send a request to <http://server1.example.com/healthtest> every 4 seconds. This probe requires that at least 3 requests succeed within a sliding window of 5 request.

Varnish initializes backends marked as sick. The variable `.initial` defines how many times the *probe* must succeed to mark the backend as healthy. `.initial` defaults to `.threshold - 1`.

The variable `Backend_health` of `varnishlog` shows the result of a backend health probe. Issue `man vs1` to see its detailed syntax.

When Varnish has no healthy backend available, it attempts to use a *graced* copy of the cached object that a request is looking for. The next section [Grace Mode](#) explains this concept in detail.

```
backend one {
    .host = "example.com";
    .probe = {
        .request =
            "GET / HTTP/1.1"
            "Host: www.foo.bar"
            "Connection: close";
    }
}
```



You can also declare standalone probes and reuse them for several backends. It is particularly useful when you use directors with identical behaviors, or when you use the same health check procedure across different web applications.

```
import directors;

probe www_probe {
    .url = "/health";
}

backend www1 {
    .host = "localhost";
    .port = "8081";
    .probe = www_probe;
}

backend www2 {
    .host = "localhost";
    .port = "8082";
    .probe = www_probe;
}

sub vcl_init {
    new www = directors.round_robin();
    www.add_backend(www1);
    www.add_backend(www2);
}
```

Note

Varnish does NOT send a Host header with health checks. If you need that, you can define an entire request using `.request` instead of `.url`.

Note

The `healthy` function is implemented as VMOD in Varnish 4. `req.backend.healthy` from Varnish 3 is replaced by `std.healthy(req.backend_hint)`. Do not forget to include the import line: `import std;`



10.3 Grace Mode

- A *graced* object is an object that has expired, but is still kept in cache.
- *Grace mode* is when Varnish uses a *graced* object.
- *Grace mode* is a feature to mitigate thread pile-ups, that allows Varnish to continue serving requests when the backend cannot do it.
- There is more than one way Varnish can use a graced object.
- `beresp.grace` defines the time that Varnish keeps an object after `beresp.ttl` has elapsed.

When Varnish is in *grace mode*, Varnish is capable of delivering a stale object and issue an asynchronous refresh request. When possible, Varnish delivers a fresh object, otherwise Varnish looks for a stale object. This procedure is also known as *stale-while-revalidate*.

The most common reason for Varnish to deliver a *graced object* is when a backend health-probe indicates a sick backend. Varnish reads the variable `obj.grace`, which default is 10 seconds, but you can change it by three means: 1) by parsing the HTTP Cache-Control field *stale-while-revalidate* that comes from the backend, 2) by setting the variable `beresp.grace` in VCL, or 3) by changing the `grace` default value with `varnishadm param.set default_grace 20`.

In the first case, Varnish parses *stale-while-revalidate* automatically, as in: `"Cache-control: max-age=5, stale-while-revalidate=30"`. In this example, the result of fetched object's variables are: `obj.ttl=5` and `obj.grace=30`. The second and third case are self descriptive.

The typical way to use *grace* is to store an object for several hours after its `TTL` has elapsed. In this way, Varnish has always a copy to be delivered immediately, while fetching a new object asynchronously. If the backend is healthy, a *graced* object does not get older than a few seconds (after its `TTL` has elapsed). If the backend is sick, Varnish may be delivering a *graced* object up to its maximum *grace* time. The following VCL code illustrates a normal usage of *grace*.

```
sub vcl_hit {
    if (obj.ttl >= 0s) {
        # Normal hit
        return (deliver);
    } elsif (std.healthy(req.backend_hint)) {
        # The backend is healthy
        # Fetch the object from the backend
        return (fetch);
    } else {
        # No fresh object and the backend is not healthy
        if (obj.ttl + obj.grace > 0s) {
            # Deliver graced object
            # Automatically triggers a background fetch
            return (deliver);
        } else {
            # No valid object to deliver
        }
    }
}
```



```
        # No healthy backend to handle request
        # Return error
        return (synth(503, "Backend is down"));
    }
}
```

Note

`obj.ttl` and `obj.grace` are countdown timers. Objects are valid in cache as long as they have a positive remaining time equal to `obj.ttl + obj.grace`.



10.3.1 Time-line example

Backend response HTTP Cache-Control header field:

```
"Cache-control: max-age=60, stale-while-revalidate=30"
```

or set in VCL:

```
set req.ttl = 60s;  
set beresp.grace = 30s;
```

- 50s: Normal delivery
- 62s: Normal cache miss, but grace mode possible
- 80s: Normal cache miss, but grace mode possible
- 92s: Normal cache miss, object is removed from cache

In this time-line example, it is assumed that the object is never refreshed.

If you do not want that objects with a negative `TTL` are delivered, set `beresp.grace = 0`. The downside of this is that all grace functionality is disabled, regardless any reason.



10.3.2 When can grace happen

- A request is already pending for some specific content
- No healthy backend is available

The main goal of *grace mode* is to avoid requests to pile up whenever a popular object has expired in cache. As long as a request is waiting for new content, Varnish delivers graced objects instead of queuing incoming requests. These requests may come from different clients, thus, large number of clients benefit from *grace mode* setups.



10.3.3 Exercise: Grace

1. Copy the following CGI script in `/usr/lib/cgi-bin/test.cgi`:

```
#!/bin/sh
sleep 10
echo "Content-type: text/plain"
echo "Cache-control: max-age=10, stale-while-revalidate=20"
echo
echo "Hello world"
date
```

2. Make the script executable.
3. Issue `varnishlog -i VCL_call,VCL_return` in one terminal.
4. Test that the script works outside Varnish by typing `http http://localhost:8080/cgi-bin/test.cgi` in another terminal.
5. Send a single request, this time via Varnish, to cache the response from the CGI script. This should take 10 seconds.
6. Send three requests: one before the TTL (10 seconds) elapses, another after 10 seconds and before 30 seconds, and a last one after 30 seconds.
7. Repeat until you understand the output of `varnishlog`.
8. Play with the values of `max-age` and `stale-while-revalidate` in the CGI script, and the `beresp.grace` value in the VCL code.

With this exercise, you should see that as long as the cached object is within its TTL, Varnish delivers the cached object as normal. Once the TTL expires, Varnish delivers the graced copy, and asynchronously fetches an object from the backend. Therefore, after 10 seconds of triggering the asynchronous fetch, an updated object is available in the cache.



10.4 `retry` return action

- Available in `vcl_backend_response` and `vcl_backend_error`
- Re-enters `vcl_backend_fetch`
- Any changes made are kept
- Parameter `max_retries` safe guards against infinite loops
- Counter `bereq.retries` registers how many retries are done

```
sub vcl_backend_response {  
    if (beresp.status == 503) {  
        return (retry);  
    }  
}
```

The `retry` return action is available in `vcl_backend_response` and `vcl_backend_error`. This action re-enters the `vcl_backend_fetch` subroutine. This only influences the backend thread, the client-side handling is not affected.

You may want to use this action when the backend fails to respond. In this way, Varnish can retry the request to a different backend. For this, you must define multiple backends.

You can use directors to let Varnish select the next backend to try. Alternatively, you may use `bereq.backend` to specifically select another backend.

`return (retry)` increments the `bereq.retries` counter. If the number of retries is higher than `max_retries`, control is passed to `vcl_backend_error`.

Note

In Varnish 3.0 it is possible to do `return (restart)` after the backend response failed. This is now called `return (retry)`, and jumps back up to `vcl_backend_fetch`.



10.5 Tune Backend Properties

```
backend default {  
    .host = "localhost";  
    .port = "80";  
    .connect_timeout = 0.5s;  
    .first_byte_timeout = 20s;  
    .between_bytes_timeout = 5s;  
    .max_connections = 50;  
}
```

If a backend has not enough resources, it might be advantageous to set `max_connections`. So that a limited number of simultaneous connections are handled by a specific backend. All backend-specific timers are available as parameters and can be overridden in VCL on a backend-specific level.

Tip

Varnish only accepts hostnames for backend servers that resolve to a maximum of one IPv4 address *and* one IPv6 address. The parameter `prefer_ipv6` defines which IP address Varnish prefer.



10.6 Access Control Lists (ACLs)

- An ACL is a list of IP addresses
- VCL programs can use ACLs to define and control the IP addresses that are allowed to *purge*, *ban*, or do any other regulated task.
- Compare with `client.ip` or `server.ip`

```
# Who is allowed to purge....
acl local {
    "localhost";          /* myself */
    "192.168.1.0"/24; /* and everyone on the local network */
    !"192.168.1.23"; /* except for the dialin router */
}

sub vcl_recv {
    if (req.method == "PURGE") {
        if (client.ip ~ local) {
            return (purge);
        } else {
            return (synth(405));
        }
    }
}
```

An Access Control List (ACL) declaration creates and initializes a named list of IP addresses and ranges, which can later be used to match client or server IP addresses. ACLs can be used for anything. They are typically used to control the IP addresses that are allowed to send `PURGE` or *ban* requests, or even to avoid the cache entirely.

You may also setup ACLs to differentiate how your Varnish servers behave. You can, for example, have a single VCL program for different Varnish servers. In this case, the VCL program evaluates `server.ip` and acts accordingly.

ACLs are fairly simple to create. A single IP address or hostname should be in quotation marks, as `"localhost"`. ACL uses the CIDR notation to specify IP addresses and their associated routing prefixes. In Varnish's ACLs the slash `"/"` character is appended outside the quoted IP address, for example `"192.168.1.0"/24`.

To exclude an IP address or range from an ACL, and exclamation mark `"!"` should precede the IP quoted address. For example `!"192.168.1.23"`. This is useful when, for example, you want to include all the IP address in a range except the gateway.



11 Content Composition

This chapter is for the webdeveloper course only

This chapter teaches you how to glue content from independent sources into one web page.

- Cookies and how to work with them
- Edge Side Includes (ESI) and how to compose a single client-visible page out of multiple objects
- Combining ESI and Cookies
- AJAX and masquerading AJAX requests through Varnish



11.1 A typical website

Most websites follow a pattern: they have easily distinguishable parts:

- A front page
- Articles or sub-pages
- A login-box or "home bar"
- Static elements, like CSS, JavaScript and graphics

To truly utilize Varnish to its full potential, start by analyzing the structure of the website. Ask yourself this:

- What makes web pages in your server different from each other?
- Does the differences apply to entire pages, or only parts of them?
- How can I let Varnish to know those differences?

Beginning with the static elements should be easy. Previous chapters of this book cover how to handle static elements. How to proceed with dynamic content?

An easy solution is to only cache content for users that are not logged in. For news-papers, that is probably enough, but not for web-shops.

Web-shops re-use objects frequently. If you can isolate the user-specific bits, like the shopping cart, you can cache the rest. You can even cache the shopping cart, if you tell Varnish when to change it.

The most important lessons is to start with what you know.



11.2 Cookies

- **Be careful when caching cookies!**

Cookies are frequently used to identify unique users, or user-choices. They can be used for anything from identifying a user-session in a web-shop to opting for a mobile version of a web page. Varnish can handle cookies coming from three different sources:

- `req.http.Cookie` header field from clients
- `beresp.http.Set-Cookie` header field from servers

By default Varnish does not cache a page if the `Cookie` request header or `Set-Cookie` response header are present. This is for two main reasons: 1) to avoid littering the cache with large amount of copies of the same content, and 2) to avoid delivering cookie-based content to a wrong client.

It is far better to either cache multiple copies of the same content for each user or cache **nothing** at all, than caching personal, confidential or private content and deliver it to a wrong client. In other words, the worst is to jeopardize users' privacy for saving backend resources. Therefore, it is strongly advised to take your time to write a correct VCL program and test it thoroughly before caching cookies in production deployments.

Despite cookie-based caching being discouraged, Varnish can be forced to cache content based on cookies. If a client request contains `req.http.Cookie`, issue `return (hash);` in `vcl_recv`. If the cookie is a `Set-Cookie` HTTP response header from the server, issue `return (deliver);` in `vcl_backend_response`.



11.2.1 *Vary and Cookies*

- Varnish **does not cache** when cookies are involved by default.
- The `Vary` response header field can be used to cache content that is based on the value of cookies.
- Cookies are widely used, but almost no-one sends `Vary: Cookie` for **content that varies based on cookies**.

Server responses containing `Vary: Cookie` in their response header are stored as a separate *vary* object in the cache. *vary* objects shared the same hash value.



11.2.2 *Best Practices for Cookies*

- Remove all cookies that you do not need
- Organize the content of your web site in a way that let you easily determine if a page needs a cookie or not. For example:
 - /common/ -- no cookies
 - /user/ -- has user-cookies
 - /voucher/ -- has only the voucher-cookie
 - etc.
- Add the `req.http.Cookie` request header to the cache hash by issuing `hash_data(req.http.cookie);` in `vcl_hash`.
- Never cache a `Set-Cookie` header. Either remove the header before caching or do not cache the object at all.
- Finish `vcl_backend_response` with something similar to:

```
if (beresp.ttl > 0s) {  
    unset beresp.http.Set-cookie;  
}
```

This ensures that all cached pages are stripped of `Set-cookie`.



11.2.3 Exercise: Compare Vary and hash_data

Both a `Vary: Cookie` response header and `hash_data(req.http.Cookie)`; create separate objects in the cache. This exercise is all about `Vary` and hash dynamics.

1. Copy the file `material/webdev/cookies.php` to `/var/www/html/cookies.php`.
2. Test `cookies.php` by issuing:

```
http -p hH http://localhost/cookies.php "Cookie: user=John"
```

3. Write a VCL program to force Varnish to cache the response from `cookies.php`.
4. Change the cookie, and see if you get a new value.
5. Make `cookies.php` send a `Vary: Cookie` header, then try changing the cookie again.
6. Try to *purge* the `cookies.php` cached object. Check if it affects all, none or just one of the objects in cache (e.g: change the value of the cookie and see if the `PURGE` method has purged all of them).
7. Remove `beresp.http.Vary` in `vcl_backend_fetch` and see if Varnish still honors the `Vary` header.
8. Add `hash_data(req.http.cookie);` in `vcl_hash`. Check how multiple values of cookie give individual cached pages.
9. Try *purging* again, and check the result difference after using `hash_data()` instead of `Vary: Cookie`.

After this exercise, you should have a very good idea on how `Vary` and `hash_data()`; work. The exercise only looks for the `Cookie` header, but the same rules apply to any other header.



11.3 Edge Side Includes

- What is ESI?
- How to use ESI?
- Testing ESI without Varnish

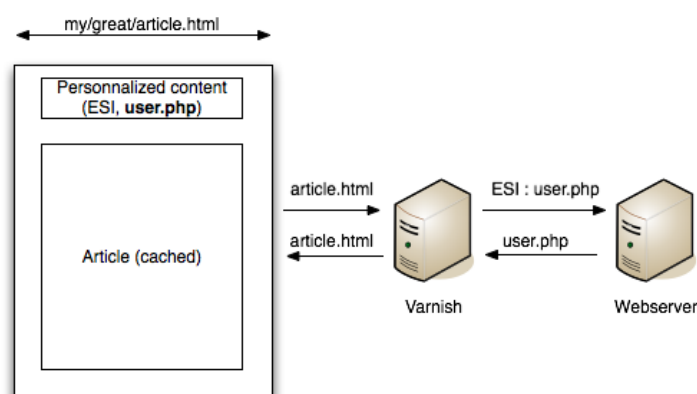


Figure 25: Web page assembling using ESI via Varnish

Edge Side Includes or ESI is a small markup language for dynamic web page assembly at the reverse proxy level. The reverse proxy analyses the HTML code, parses ESI specific markup and assembles the final result before flushing it to the client. [Figure 25](#) depicts this process.

With ESI, Varnish can be used not only to deliver objects, but to glue them together. The most typical use case for ESI is a news article with a most recent news box at the side. The article itself is most likely written once and possibly never changed, and can be cached for a long time. The box at the side with most recent news, however, changes frequently. With ESI, the article can include a most recent news box with a different TTL.

When using ESI, Varnish fetches the news article from a web server, then parses the `<esi:include src="/url" />` ESI tag, and fetches the URL via a normal request. Either finding it already cached or getting it from a web server and inserting it into cache.

The TTL of the ESI element can be 5 minutes while the article is cached for two days. Varnish delivers the two different objects in one glued page. Thus, Varnish updates parts independently and makes possible to combine content with different TTL.



11.3.1 Basic ESI usage

Enabling ESI in Varnish is simple enough:

```
sub vcl_backend_response {  
    set beresp.do_esi = true;  
}
```

To include a page in another, the `<esi:include>` ESI tag is used:

```
<esi:include src="/url" />
```

You can also strip off cookies per ESI element. This is done in `vcl_recv`.

Varnish only supports three ESI tags:

- `<esi:include>`: calls the page defined in the `src` attribute and replaces the ESI tag with the content of `src`.
- `<esi:remove>`: removes any code inside this opening and closing tag.
- `<!--esi `(content) -->``: Leaves (content) unparsed. E.g., the following does not process the `<esi:include>` tag:

```
<!--esi  
    This ESI tag is not processed: <esi:include src="example">  
-->
```

Note

Varnish outputs ESI parsing errors in `varnishstat` and `varnishlog`.



11.3.2 Example: Using ESI

Copy `material/webdev/esi-date.php` to `/var/www/html/`. This file contains an ESI include tag.

```
<HTML>
<BODY>

<?php
header( 'Content-Type: text/plain' );

print( "This page is cached for 1 minute.\n" );
echo "Timestamp: \n"
. date("Y-m-d H:i:s");
print( "\n" );
?>

<esi:include src="/cgi-bin/date.cgi"/>

</BODY>
</HTML>
```

Copy `material/webdev/esi-date.cgi` to `/usr/lib/cgi-bin/`. This file is a simple CGI that outputs the date of the server.

```
#!/bin/sh

echo "Content-Type: text/plain"
echo ""
echo "ESI content is cached for 30 seconds."
echo "Timestamp: "
date "+%Y-%m-%d %H:%M:%S"
```

For the ESI to work, load the following VCL code.

```
sub vcl_backend_response {
    if (bereq.url == "/esi-date.php") {
        set beresp.do_esi = true;    // Do ESI processing
        set beresp.ttl = 1m;        // Sets a higher TTL main object
    } elseif (bereq.url == "/cgi-bin/esi-date.cgi") {
        set beresp.ttl = 30s;        // Sets a lower TTL on
                                    // the included object
    }
}
```

Then reload Varnish and issue the command `http http://localhost/esi-date.php`. The output should show you how Varnish replaces the ESI tag with the response from `esi-date.cgi`. This example also tries to show you how the glued objects have different TTLs.



11.3.3 Exercise: Enable ESI and Cookies

1. Use `material/webdev/esi-top.php` and `material/webdev/esi-user.php` to test ESI.
2. Visit `esi-top.php` and identify the ESI tag.
3. Enable ESI for `esi-top.php` in VCL and test.
4. Strip all cookies from `esi-top.php` and make it cache.
5. Let `esi-user.php` cache too. It emits `Vary: Cookie`, but might need some help.

Try to avoid `return (hash);` in `vcl_recv` and `return (deliver);` in `vcl_backend_response` as much as you can. This is a general rule to make safer Varnish setups.

During the exercise, make sure you understand all the cache mechanisms at play. You can also try removing the `Vary: Cookie` header from `esi-user.php`.

You may also want to try `PURGE`. If so, you have to purge each of the objects, because purging just `/esi-top.php` does not purge `/esi-user.php`.



11.3.4 Testing ESI without Varnish

- Test ESI Using JavaScript to fill in the blanks.

During development of different web pages to be ESI-glued by Varnish, you might not need Varnish all the time. One important reason for this, is to avoid caching during the development phase. There is a solution based on JavaScript to interpret ESI syntax without having to use Varnish at all. You can download the library at the following URL:

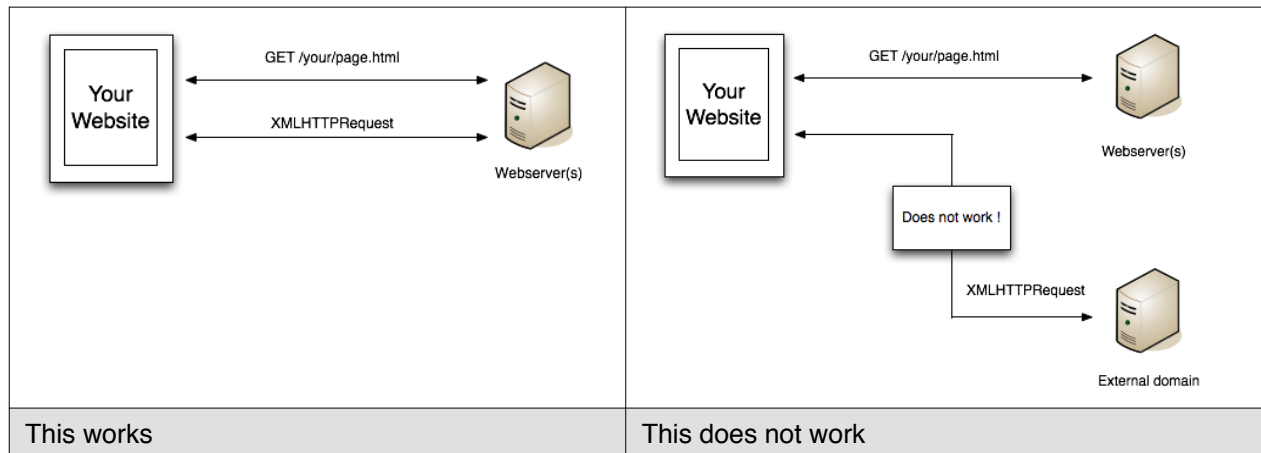
- http://www.catalystframework.org/calendar/static/2008/esi/ESI_Parser.tar.gz

Once downloaded, extract it in your code base, include `esiparser.js` and include the following JavaScript code to trigger the ESI parser:

```
$(document).ready( function () { do_esi_parsing(document); });
```



11.4 Masquerading AJAX requests



With AJAX it is not possible by default to send requests across another domain. This is a security restriction imposed by browsers. If this represents an issue for your web pages, you can be easily solve it by using Varnish and VCL.



11.4.1 Exercise: write a VCL that masquerades XHR calls

material/webdev/ajax.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.4/jquery.min.js">
    </script>
    <script type="text/javascript">
      function getNonMasqueraded()
      {
        $("#result").load( "http://www.google.com/robots.txt" );
      }

      function getMasqueraded()
      {
        $("#result").load( "/masq/robots.txt" );
      }
    </script>
  </head>
  <body>
    <h1>Cross-domain Ajax</h1>
    <ul>
      <li><a href="javascript:getNonMasqueraded();">
        Test a non masqueraded cross-domain request
      </a></li>
      <li><a href="javascript:getMasqueraded();">
        Test a masqueraded cross-domain request
      </a></li>
    </ul>

    <h1>Result</h1>
    <div id="result"></div>
  </body>
</html>
```

Use the provided ajax.html page. Notice that function `getNonMasqueraded()` fails because the origin is distinct to the `google.com` domain. Function `getMasqueraded()` can do the job if a proper VCL code handles it. Write the VCL code that masquerades the Ajax request to `http://www.google.com/robots.txt`.



11.4.2 Solution: write a VCL that masquerades XHR calls`vcl/solution-vcl_fetch-masquerade-ajax-requests.vcl`

```
vcl 4.0;

backend localhost{
    .host = "127.0.0.1";
    .port = "8080";
}

backend google {
    .host = "173.194.112.145";
    .port = "80";
}

sub vcl_recv{
    if (req.url ~ "^/masq") {
        set req.backend_hint = google;
        set req.http.host = "www.google.com";
        set req.url = regsub(req.url, "^/masq", "");
        return (hash);
    } else {
        set req.backend_hint = localhost;
    }
}
```

Notice that the `getMasqueraded()` works now after being processed in `vcl_recv()`.



12 Varnish Plus Software Components

The Varnish Plus offer of software products includes:

- Varnish Massive Storage Engine (MSE), described in the [Storage Backends](#) section.
- Hashtwo (Varnish Software Implementation of Surrogate Keys)
- Varnish Tuner
- Varnish Administration Console (VAC),
- Varnish Custom Statistics (VCS),
- Varnish High Availability (VHA),
- SSL/TLS Support,
- and more.

For more information about the complete Varnish Plus offer and their documentation, please visit:

- <https://www.varnish-software.com/what-is-varnish-plus>
- <https://www.varnish-software.com/resources/>



12.1 Varnish Administration Console (VAC)

- Single point of control for simultaneous administration of multiple Varnish Cache servers
- VAC provides:
 - GUI
 - API
 - Super Fast Purger
- VAC has its own documentation

The Varnish Administration Console (VAC) consists of a GUI and an API. VAC is most commonly used in production environments where real-time graphs and statistics help identify bottlenecks and issues within Varnish Cache servers. VAC is a management console for groups of Varnish Cache servers, also known as cache groups. A cache group is a collection of Varnish Cache servers that have identical configuration. Attributes on a cache group includes:

- One or more Varnish Cache servers
- Active Varnish Configuration Language (VCL) file
- Homogeneous parameter configuration across all servers in a group

VAC distributes and store VCL files for you. A parameter set is a list of Varnish cache parameters. These parameters can be applied to one or more cache groups simultaneously, as long as all cache groups consist of cache servers of the same version.

VAC ships with a JSON-based RESTful API to integrate your own systems with the VAC. All actions performed via the user interface can be replicated with direct access to the API. This includes fetching all real-time graph data.

The Super Fast Purger is a high performance cache invalidation delivery mechanism for multiple installations of Varnish. Super Fast Purger is capable of distributing purge requests to cache groups across data centers via the Restful interface. Super Fast Purger uses HMAC as security mechanism to protect your purge requests and thus ensure data integrity.

In order to install VAC on either Debian/Ubuntu or Red Hat Enterprise, one would require access to the Varnish Plus Software repository. As a Varnish Plus customer, you have access to the installation guide document. This document has instructions to install, configure, maintain, and troubleshoot your VAC installation. If you have any questions on how to set up your repository or where to obtain the VAC installation guide, please ask the instructor or send us an email to support@varnish-software.com.

Figures 26, 27 and 28 show screenshots of the GUI. You may also be interested in trying the VAC demo at <https://vacdemo.varnish-software.com>. The instructor of the course provides you the credentials.



12.1.1 Overview Page of the Varnish Administration Console

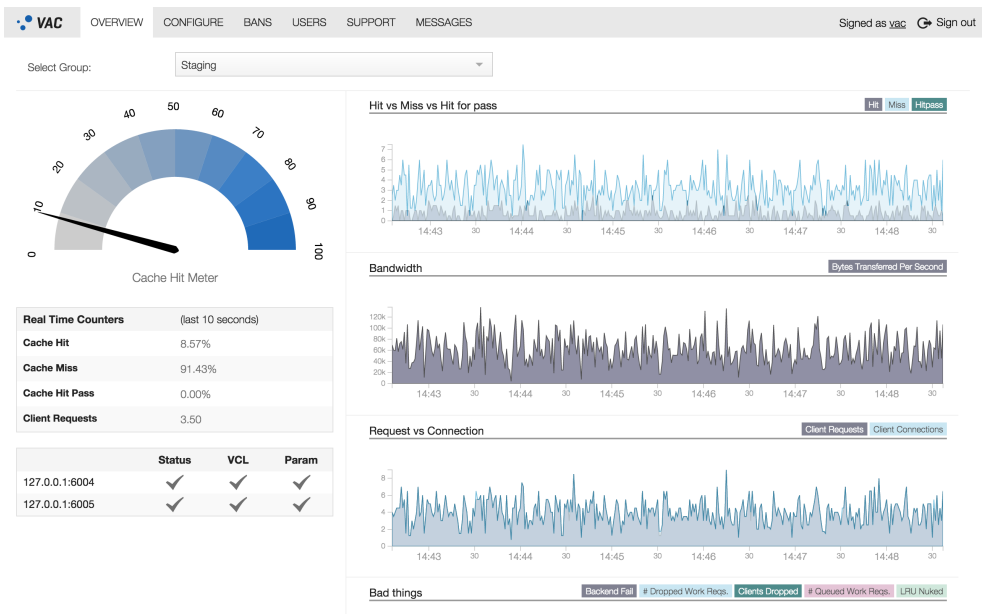


Figure 26: Overview page of the Varnish Administration Console



12.1.2 Configuration Page of the Varnish Administration Console

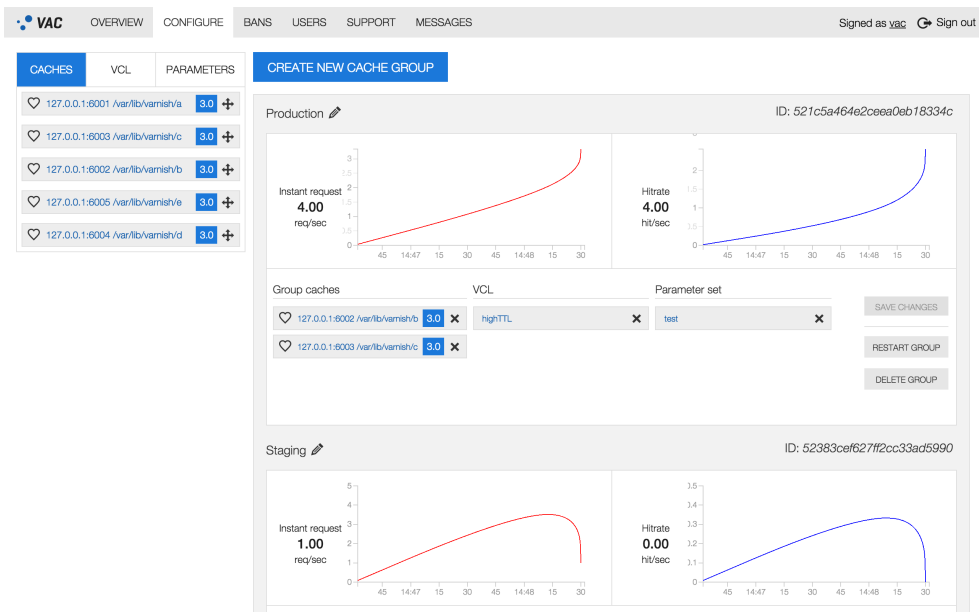


Figure 27: Configuration page of the Varnish Administration Console



12.1.3 Banning Page of the Varnish Administration Console

VACOVERVIEWCONFIGUREBANUSERSUPPORTMESSAGES

Signed as vacSign out

BAN EXPRESSION:

GROUP:

Staging

EXECUTE BAN

Recent Bans

Expression	Group	Date & Time
req.url ~ /	Staging	19/11/2014 17:51:58
req.url ~ /	Staging	13/11/2014 23:02:58
req.http.host ~ ".*\example.com" && req.url ~ ".*\.png\$"	Staging	12/11/2014 13:03:06
req.url ~ /	Staging	10/11/2014 13:39:41
req.url ~ /	Staging	07/11/2014 14:19:59
req.url ~ /	Staging	07/11/2014 14:19:49
req.url ~ /	Staging	07/11/2014 14:19:41
req.url ~ /	Staging	07/11/2014 14:17:35
req.url ~ /	Staging	22/10/2014 20:15:01

LOAD MORE BANS

Bans

Bans is how you invalidate content currently in the cache.

In this view you can add bans that are executed on all the caches in a cache group. The ban text is the complete expression as it would be handed to ban() in VCL.

Click on one of the examples below to use:

Ban Expression	Description
req.url ~ /	

Figure 28: Banning page of the Varnish Administration Console



12.2 Varnish Custom Statistics (VCS)

- Data stream management system (DSMS) for your Varnish servers
- Real-time statistics engine to aggregate, display and analyze user web traffic
- Provides an API to retrieve statistics
- Provides a GUI that presents lists and charts to get a quick overview of the key metrics that matters you

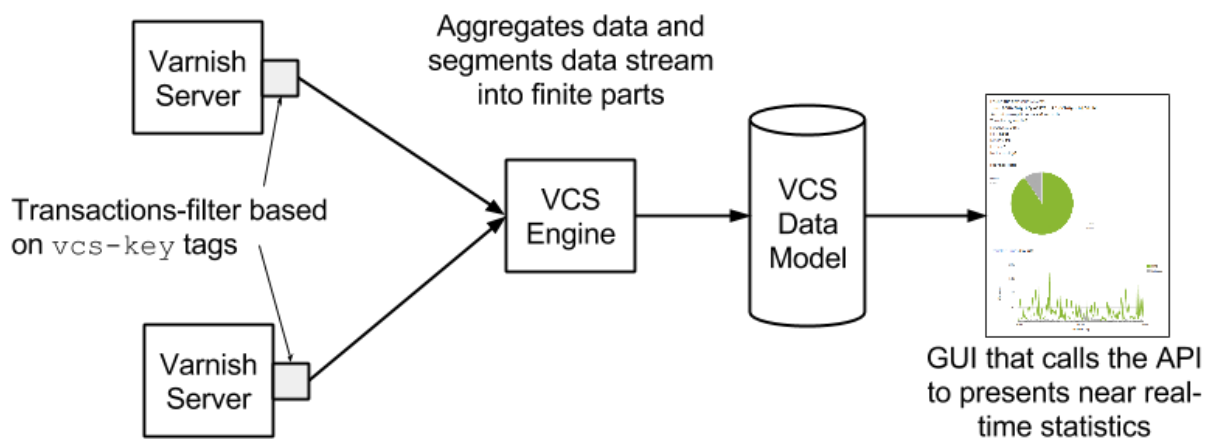


Figure 29: VCS Data Flow

Varnish Custom Statistics (VCS) is our data stream management system (DSMS) implementation for Varnish. VCS allows you to analyze the traffic from multiple Varnish servers in near real-time to compute traffic statistics and detect critical conditions. This is possible by continuously extracting transactions with the `vcs-key` tags in your VSL. Thus, VCS does not slow down your Varnish servers.

You can add as many custom `vcs-key` tags as you need in your VCL code. This allows you to define your own metrics.

VCS can be used to produce statistical data or even apply complex event processing techniques. Thus, VCS offers endless opportunities for tracking all aspects of websites' behavior. Typical cases include:

- A/B testing
- Measuring click-through rate
- Track slow pages and cache misses
- Analyze what is "hot" right now in a news website
- Track changes in currency conversions in e-commerce
- Track changes in Stock Keeping Units (SKUs) <behavior in e-commerce
- Track number of unique consumers of HLS/HDS/DASH video streams



Figure 30 and 31 are screenshots of the VCS GUI. These screenshots are from the demo on <http://vcsdemo.varnish-software.com>. Your instructor can provide you credential for you to try the demo online.

Note

For further details on VCS, please look at its own documentation at <https://www.varnish-software.com/resources/>.



12.2.1 VCS Data Model

- Represents a finite relation from an infinite stream.
- Uses time-based tumbling windows
- API to query data model
- API outputs in JSON and JSONP format

Table 20: Data model in VCS

vcs-key	example.com	example.com
timestamp	2013-09-18T09:58:00	2013-09-18T09:58:30
n_req	84	76
n_req_uniq	NaN	NaN
n_miss	0	1
avg_restart	0.000000	0.000000
n_bodybytes	12264	10950
ttfb_miss	NaN	0.000440
ttfb_hit	0.000048	0.000054
resp_1xx	0	0
resp_2xx	84	76
resp_3xx	0	0
resp_4xx	0	0
resp_5xx	0	0
reqbytes	8	6
respbytes	32	29
berespbytes	30	27
bereqbytes	9	7

VCS uses the time-based tumbling windows technique to segment the data stream into finite parts. These windows are created based on the `vcs-key` tag that you specify in your VCL code. Each window aggregates the data within a configurable period of time.

Table 20 shows the data model in VCS. This table is basically a representation of two windows seen as two records in a conventional database. In this example, data shows two windows of 30 second based on the `example.com vcs-key`. For presentation purposes in this page, the distribution of this table is of a database that grows from left to right.

The VCS data model has the following fields:



vcs-key

common key name for transactions making this record

timestamp

Timestamp at the start of the window

n_req

Number of requests

n_req_uniq

Number of unique requests, if configured

n_miss

Number of backend requests (i.e. cache misses) Number of hits can be calculated as
 $n_hit = n_req - n_miss$

avg_restart

Average number of VCL restarts triggered per request

n_bodybytes

Total number of bytes transferred for the response bodies

ttfb_miss

Average time to first byte for requests that ended up with a backend request

ttb_hit

Average time to first byte for requests that were served directly from varnish cache

resp_1xx -- resp_5xx

Counters for response status codes.

reqbytes

Number of bytes received from clients.

respbytes

Number of bytes transmitted to clients.

berespbytes

Number of bytes received from backends.

bereqbytes

Number of bytes transmitted to backends.

You can think of each window as a record of a traditional database that resides in memory. This database is dynamic, since the engine of VCS updates it every time a new window (record) is available. VCS provides an API to retrieve this data from the table above in JSON format:

```
{
  "example.com": [
    {
      "timestamp": "2013-09-18T09:58:30",
      "n_req": 76,
      "n_req_uniq": "NaN",
      "n_miss": 1,

```



```
    "avg_restarts": 0.000000,  
    "n_bodybytes": 10950,  
    "ttfb_miss": 0.000440,  
    "ttfb_hit": 0.000054,  
    "resp_1xx": 0,  
    "resp_2xx": 76,  
    "resp_3xx": 0,  
    "resp_4xx": 0,  
    "resp_5xx": 0,  
    ...  
  },  
  {  
    "timestamp": "2013-09-18T09:58:00",  
    "n_req": 84,  
    "n_req_uniq": "NaN",  
    "n_miss": 0,  
    "avg_restarts": 0.000000,  
    "n_bodybytes": 12264,  
    "ttfb_miss": "NaN",  
    "ttfb_hit": 0.000048,  
    "resp_1xx": 0,  
    "resp_2xx": 84,  
    "resp_3xx": 0,  
    "resp_4xx": 0,  
    "resp_5xx": 0,  
    ...  
  },  
  ...  
]  
}
```



12.2.2 VCS API

- API provides ready-to-use queries
- Queries over HTTP
- Top most sorting
- Results in JSON and JSONP format

Examples:

For `vcs-key` with names ending with `.gif`, retrieve a list of the top 10:

```
/match/(.*)%5C.gif$/top
```

Find a list of the top 50 slowest backend requests:

```
/all/top_ttfb/50
```

The VCS API queries the VCS data model and the output is in JSON format. The API responds to requests for the following URLs:

/key/<vcs-key>

Retrieves stats for a single `vcs-key`. `<vcs-key>` name must be URL encoded.

/match/<regex>

Retrieves a list of `vcs-key` matching the URL encoded regular-expression. Accepts the query parameter `verbose=1`, which displays all stats collected for the `<vcs-keys>` matched.

/all

Retrieves a list of all the `<vcs-keys>` currently in the data model.

For `/match/<regex>` and `/all`, VCS can produce sorted lists. For that, you can append one of the following sorting commands.

/top

Sort based on number of requests.

/top_ttfb

Sort based on the `ttfb_miss` field.

/top_size

Sort based on the `n_bodybytes` field.

/top_miss

Sort based on the `n_miss` field.

/top_respbytes

Sort based on number of bytes transmitted to clients.

/top_reqbytes

Sort based on number of bytes received from clients.



/top_berespbytes

Sort based on number of bytes fetched from backends.

/top_bereqbytes

Sort based on number of bytes transmitted to backends.

/top_restarts

Sort based on the `avg_restarts` field.

/top_5xx, /top_4xx, ..., /top_1xx

Sort based on number of HTTP response codes returned to clients for 5xx, 4xx, 3xx, etc.

/top_uniq

Sort based on the `n_req_uniq` field.

Further, a `/k` parameter can be appended, which specifies the number of keys to include in the top list. If no `k` value is provided, the top 10 is displayed.

Note

For more details on the API, please read the documentation of `vstatd`.



12.2.3 Screenshots of GUI

- VCS provides its own GUI
- You can interact with the API via this GUI
- Screenshots from <http://vcsdemo.varnish-software.com/>



Figure 30: Header of Varnish Custom Statistics

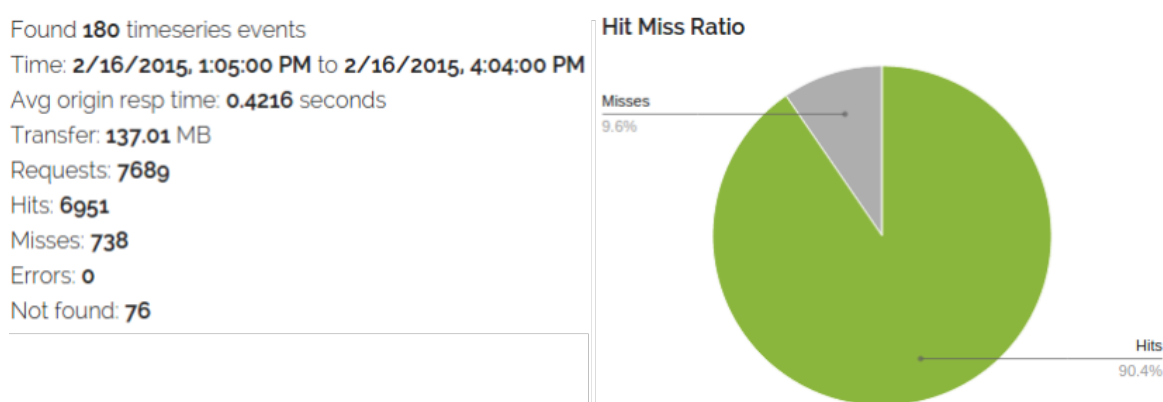


Figure 31: Summary of metrics along with time based graphs



12.3 Varnish High Availability (VHA)

- Content replicator
- Increases resiliency and performance
- Two-server, circular, multi-master replication
- Requests to replicate content against Varnish servers, not the backend

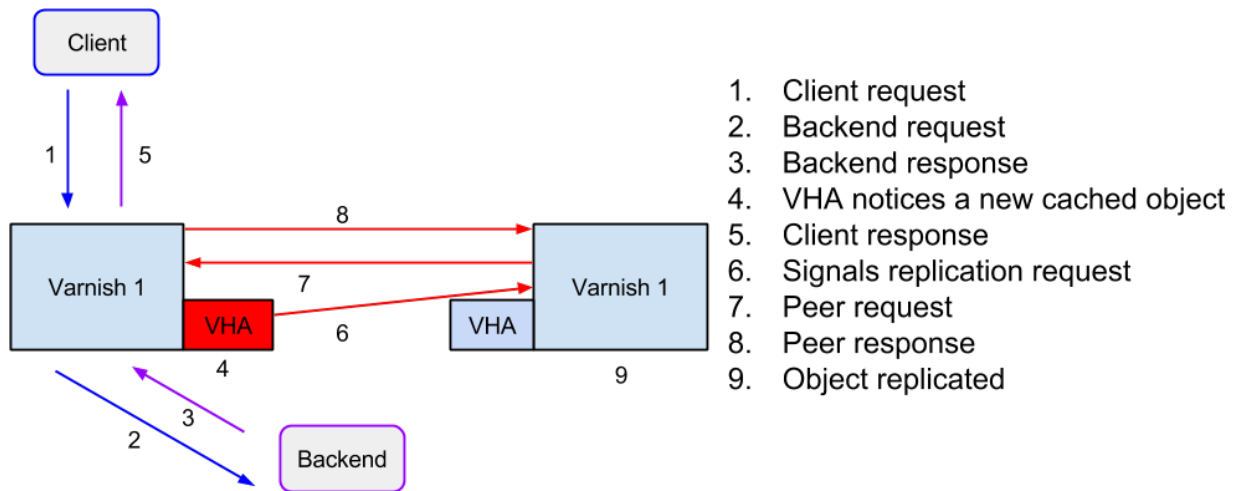


Figure 32: VHA Sequence Diagram

The Varnish High Availability agent (*vha-agent*) is a content replicator with the aim of copying the cached objects from an origin Varnish server to a neighboring Varnish server. This increases resiliency and performance, specially when backend traffic surges.

vha-agent reads the log of Varnish, and for each object insertion detected it fires a request to the neighboring Varnish server. This server fetches the object from the origin Varnish server. As a result, the same object is cached in both servers with only one single backend fetch.

This solution requires *vha-agent* to be installed on the origin Varnish server, and some simple VCL configuration on the replicated Varnish server. Ideally, *vha-agent* is installed on both servers so they can both replicate object insertions from each other in an active/active configuration.

Typical uses of VHA include:

- Business critical Varnish Plus installations
- Any multi-cache Varnish setup
- Multi node CDN POP installations

The replication of cached objects may bring the need for multiple cache invalidation. For that purpose, you can use the [Varnish Administration Console \(VAC\)](#). Remember: you should define the rules on how to invalidate cached objects before caching them in production environments.



12.4 SSL/TLS Support

- SSL/TLS support on both the HTTP backend and client side.
- Easy configuration:

```
backend default {  
    .host = "host.name";  
    .port = "https";  
    .ssl = 1; # This defaults to https when SSL  
    .ssl_nosni = 1; # Turns on SSL support  
    .ssl_noverify = 1; # Disable SNI extension  
                # Don't verify peer  
}
```

Varnish Plus allows you to improve your website security without having to rely on third-party solutions. SSL/TLS support allows you to encrypt and secure communication on both the front and backend as Varnish acts as both HTTP server and client. On the client side, the HTTP server intercepts web requests before they reach a web server. The SSL/TLS support on this side enables traffic encryption between the client and Varnish.

On the backend, the HTTP client fetches content missing in the cache from the web server. This enables content to be fetched over the encrypted SSL/TLS, which particularly benefits customers who run a fully encrypted data center or have web servers that reside in a different location to their Varnish Plus servers.



13 Appendix A: Resources

Community driven:

- <https://www.varnish-cache.org>
- <https://www.varnish-cache.org/docs/>
- <http://repo.varnish-cache.org/>
- <https://www.varnish-cache.org/trac/wiki/VCLExamples>
- Public mailing lists: <https://www.varnish-cache.org/trac/wiki/MailingLists>
- Public IRC channel: #varnish at irc.linpro.no

Commercial:

- <https://www.varnish-software.com/resources/>
- <http://planet.varnish-cache.org/>
- <https://www.varnish-software.com>
- <http://repo.varnish-software.com> (for service agreement customers)
- support@varnish-software.com (for existing customers, with SLA)
- sales@varnish-software.com



14 Appendix B: Varnish Programs

SHared Memory LOG (SHMLOG) tools:

- `varnishlog`
- `varnishncsa`
- `varnishhist`
- `varnishtop`

Administration:

- `varnishadm`

Global counters:

- `varnishstat`

Misc:

- `varnishtest`

`varnishlog`, `varnishadm` and `varnishstat` are explained in the [Examining Varnish Server's Output](#) chapter. `varnishtest` is used for regression tests, mainly during development, but it also useful to learn more about Varnish's behavior. For more information about `varnishtest`, see its man page.

Next sections explain `varnishtop`, `varnishncsa`, and `varnishhist`.



14.1 varnishtop

```
$varnishtop -i BereqURL,RespStatus

list length 5                                     trusty-amd64

    7.20 RespStatus      200
    5.26 RespStatus      404
    0.86 BereqURL        /test.html
    0.68 BereqURL        /
    0.39 BereqURL        /index.html
```

- Group tags and tag-content by frequency

`varnishtop` groups tags and their content together to generate a sorted list of the most frequently appearing tag/tag-content pair. This tool is sometimes overlooked, because its usefulness is visible after you start filtering. The above example lists status codes that Varnish returns.

Two of the perhaps most useful variants of `varnishtop` are:

- `varnishtop -i BereqURL`: creates a list of URLs requested at the backend. Use this to find out which URL is the most requested.
- `varnishtop -i RespStatus`: lists what status codes Varnish returns to clients.

You may also combine `taglist` as in the above example. Even more, you may apply [Query Language](#) `-q` options. For example, `varnishtop -q 'respstatus > 400'` shows you counters for responses where client seem to have erred.

Some other possibly useful examples are:

- `varnishtop -i ReqUrl`: displays what URLs are most frequently requested from clients.
- `varnishtop -i ReqHeader -I 'User-Agent:.*Linux.*'`: lists User-Agent headers with "Linux" in it. This example is useful for Linux users, since most web browsers in Linux report themselves as Linux.
- `varnishtop -i RespStatus`: lists status codes received in clients from backends.
- `varnishtop -i VCL_call`: shows what VCL functions are used.
- `varnishtop -i ReqHeader -I Referrer` shows the most common referrer addresses.



14.2 varnishncsa

```
10.10.0.1 - - [24/Aug/2008:03:46:48 +0100] "GET \
http://www.example.com/images/foo.png HTTP/1.1" 200 5330 \
"http://www.example.com/" "Mozilla/5.0"
```

If you already have tools in place to analyze NCSA Common log format, `varnishncsa` can be used to print the SHMLOG in this format. `varnishncsa` dumps everything pointing to a certain domain and subdomains.

Filtering works similar to `varnishlog`.



14.4 Exercise: Try the tools

- Send a few requests to Varnish using `http -p hH http://localhost/`
- verify you have some cached objects using `varnishstat`
- look at the communication with the clients, using `varnishlog`. Try sending various headers and see them appear in `varnishlog`.
- Install `siege`
- Run `siege` against `localhost` while looking at `varnishhist`



15 Appendix C: Extra Material

The following is content needed for some of the exercises.



15.1 ajax.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript"
      src="http://ajax.googleapis.com/ajax/libs/jquery/1.4/jquery.min.js">
    </script>
    <script type="text/javascript">
      function getNonMasqueraded()
      {
        $("#result").load( "http://www.google.com/robots.txt" );
      }

      function getMasqueraded()
      {
        $("#result").load( "/masq/robots.txt" );
      }
    </script>
  </head>
  <body>
    <h1>Cross-domain Ajax</h1>
    <ul>
      <li><a href="javascript:getNonMasqueraded();">
        Test a non masqueraded cross-domain request
      </a></li>
      <li><a href="javascript:getMasqueraded();">
        Test a masqueraded cross-domain request
      </a></li>
    </ul>

    <h1>Result</h1>
    <div id="result"></div>
  </body>
</html>
```



15.2 article.php

```
<?php
header("Cache-Control: public, must-revalidate, max-age=3600, s-maxage=3600");
$date = new DateTime();
$now = $date->format( DateTime::RFC2822 );
?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h1>This is an article, cached for 1 hour</h1>

    <h2>Now is <?php echo $now; ?></h2>
    <a href="<?=$_SERVER['PHP_SELF']?>">Refresh this page</a>
  </body>
</html>
```



15.3 cookies.php

```
<?php
header( 'Content-Type: text/plain' );

print( "The following cookies have been received from the server\n" );

foreach( $_COOKIE as $name => $value )
    print( "- ${name} : ${value}\n" );
?>
```



15.4 esi-top.php

```

<?php
header('Content-Type: text/html');
header('Cache-Control: max-age=30, s-maxage=3600');
$date = new DateTime();
$now = $date->format( DateTime::RFC2822 );
$setc = "";
if( isset($_POST['k']) and $_POST['k'] !== '' and
    isset($_POST['v']) and $_POST['v'] !== '' ) {
    $k=$_POST['k'];
    $v=$_POST['v'];
    $setc = "Set-Cookie: $k=$v";

    header("$setc");
    ?><meta http-equiv="refresh" content="1" />
    <h1>Refreshing to set cookie <?php print $setc; ?></h1><?php
}
?>
<html><head><title>ESI top page</title></head><body><h1>ESI Test page</h1>
<p>This is content on the top-page of the ESI page.
The top page is cached for 1 hour in Varnish,
but only 30 seconds on the client.</p>
<p>The time when the top-element was created:</p><h3>

<?php echo "$now"; ?>

<h1>Set a cookie:</h1><form action="/esi-top.php" method="POST">
Key: <input type="text" name="k">
Value: <input type="text" name="v">
<input type="submit"> </form>

</h3><p>The top page received the following Cookies:</p><ul>

<?php
foreach( $_COOKIE as $name => $value )
    print( "<li>${name} : ${value}</li>\n" );
?>

<table border="1"><tr><td><esi:include src="/esi-user.php" /></td></tr>
</table></body></html>

```



15.5 esi-user.php

```
<?php
header('Content-Type: text/html');
header('Cache-Control: max-age=30, s-maxage=20');
header('Vary: Cookie');
$date = new DateTime();
$now = $date->format( DateTime::RFC2822 );
?>
<p>This is content on the user-specific ESI-include. This part of
the page is can be cached in Varnish separately since it emits
a "Vary: Cookie"-header. We can not affect the client-cache of
this sub-page, since that is determined by the cache-control
headers on the top-element.</p>
<p>The time when the user-specific-element was created:</p><h3>

<?php echo "$now"; ?>

</h3><p>The user-specific page received the following Cookies:
</p><ul>

<?php
foreach( $_COOKIE as $name => $value )
    print( "<li>${name} : ${value}</li>\n" );
?>

</ul>
```





15.6 httpheadersexample.php

```
<?php
define( 'LAST_MODIFIED_STRING', 'Sat, 09 Sep 2000 22:00:00 GMT' );

// expires_date : 10s after page generation
$expires_date = new DateTime();
$expires_date->add(new DateInterval('PT10S'));

$headers = array(
    'Date' => date( 'D, d M Y H:i:s', time() ),
);

if( isset( $_GET['h'] ) and $_GET['h'] !== '' )
{
    switch( $_GET['h'] )
    {
        case "expires" :
            $headers['Expires'] = toUTCDate($expires_date);
            break;

        case "cache-control":
            $headers['Cache-Control'] = "public, must-revalidate,
            max-age=3600, s-maxage=3600";
            break;

        case "cache-control-override":
            $headers['Expires'] = toUTCDate($expires_date);
            $headers['Cache-Control'] = "public, must-revalidate,
            max-age=2, s-maxage=2";
            break;

        case "last-modified":
            $headers['Last-Modified'] = LAST_MODIFIED_STRING;
            $headers['Etag'] = md5( 12345 );

            if( isset( $_SERVER['HTTP_IF_MODIFIED_SINCE'] ) and
                $_SERVER['HTTP_IF_MODIFIED_SINCE'] ==
                LAST_MODIFIED_STRING ) {
                header( "HTTP/1.1 304 Not Modified" );
                exit( );
            }
            break;

        case "vary":
            $headers['Expires'] = toUTCDate($expires_date);
            $headers['Vary'] = 'User-Agent';
```



```

        break;
    }

    sendHeaders( $headers );
}

function sendHeaders( array $headerList )
{
    foreach( $headerList as $name => $value )
    {
        header( "${name}: ${value}" );
    }
}

function toUTCDate( DateTime $date )
{
    $date->setTimezone( new DateTimeZone( 'UTC' ) );
    return $date->format( 'D, d M Y H:i:s \G\M\T' );
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head></head>
    <body>
        <h1>Headers sent</h1>
        <?php
            foreach( $headers as $name => $value ) {
                print "<strong>${name}</strong>: ${value}<br/>";
            }

            if( isset( $_SERVER['HTTP_IF_MODIFIED_SINCE'] ) ) {
                print "<strong>If-Modified-Since</strong> has been
                sent in the";
                print "request, value : " .
                $_SERVER['HTTP_IF_MODIFIED_SINCE'];
            }
        ?>
        <hr/>
        <h1>Links for testing</h1>
        <ul>
            <li><a href="<?=$_SERVER['PHP_SELF']>?h=expires">
                Test Expires response header</a></li>
            <li><a href="<?=$_SERVER['PHP_SELF']>?h=cache-control">
                Test Cache-Control response header</a></li>
            <li><a href="<?=$_SERVER['PHP_SELF']>?h=cache-control-override">
                Test Cache-Control response header overrides Expires</a></li>
        </ul>
    </body>
</html>

```



```
<li><a href="<?=$_SERVER['PHP_SELF']?>?h=last-modified">
Test Last-Modified/If-modified-since response header</a></li>
<li><a href="<?=$_SERVER['PHP_SELF']?>?h=vary">
Test Vary response header</a></li>
<ul>
</body>
</html>
```



15.7 purgearticle.php

```
<?php
header( 'Content-Type: text/plain' );
header( 'Cache-Control: max-age=0' );
$hostname = 'localhost';
$port     = 80;
$URL      = '/article.php';
$debug    = true;

print "Updating the article in the database ...\n";
purgeURL( $hostname, $port, $URL, $debug );

function purgeURL( $hostname, $port, $purgeURL, $debug )
{
    $finalURL = sprintf(
        "http://%s:%d%s", $hostname, $port, $purgeURL
    );

    print( "Purging ${finalURL}\n" );

    $curlOptionList = array(
        CURLOPT_RETURNTRANSFER => true,
        CURLOPT_CUSTOMREQUEST  => 'PURGE',
        CURLOPT_HEADER          => true ,
        CURLOPT_NOBODY          => true,
        CURLOPT_URL              => $finalURL,
        CURLOPT_CONNECTTIMEOUT_MS => 2000
    );

    $fd = false;
    if( $debug == true ) {
        print "\n---- Curl debug ----\n";
        $fd = fopen("php://output", 'w+');
        $curlOptionList[CURLOPT_VERBOSE] = true;
        $curlOptionList[CURLOPT_STDERR]  = $fd;
    }

    $curlHandler = curl_init();
    curl_setopt_array( $curlHandler, $curlOptionList );
    curl_exec( $curlHandler );
    curl_close( $curlHandler );
    if( $fd !== false ) {
        fclose( $fd );
    }
}
?>
```



15.8 test.php

```
<?php
$cc = "";
if( isset($_GET['k']) and $_GET['k'] !== '' and
    isset($_GET['v']) and $_GET['v'] !== '' ) {
    $k=$_GET['k'];
    $v=$_GET['v'];
    $cc = "Cache-Control: $k=$v";

    header("$cc");

}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h1>Cache-Control Header:</h1>
    <?php
        print "<pre>$cc</pre>\n";
    ?>
    <hr/>
    <h1>Links for testing</h1>
    <form action="/test.php" method="GET">
      Key: <input type="text" name="k">
      Value: <input type="text" name="v">
      <input type="submit">
    </form>
  </body>
</html>
```



15.9 set-cookie.php

```
<?php
header("Cache-Control: max-age=0");
$cc = "";
if( isset($_POST['k']) and $_POST['k'] !== '' and
    isset($_POST['v']) and $_POST['v'] !== '' ) {
    $k=$_POST['k'];
    $v=$_POST['v'];
    $setc = "Set-Cookie: $k=$v";

    header("$setc");
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head></head>
  <body>
    <h1>Set-Cookie Header:</h1>
    <?php
        print "<pre>$setc</pre>\n";
    ?>
    <hr/>
    <h1>Links for testing</h1>
    <form action="/set-cookie.php" method="POST">
      Key: <input type="text" name="k">
      Value: <input type="text" name="v">
      <input type="submit">
    </form>
  </body>
</html>
```



15.10 VCL Migrator from Varnish 3 to Varnish 4

varnish3to4 is a script to assist you migrating a VCL file from Varnish 3 to 4. The script aims to replace most of the syntactical changes in VCL code from Varnish 3 to Varnish 4, but it is not exhaustive. That said, you should use it under your own responsibility.

You can download the script from <https://github.com/fgsch/varnish3to4>. Usage and up-to-date details about the script is at the same web address.



16 Appendix D: Varnish Three Letter Acronyms

VAV

Varnish Arg Vector -- Argv parsing.

VBE

Varnish Back End -- Code for contacting backends (bin/varnishd/cache_backend.c)

VBP

Varnish Backend Polling -- Health checks of backends (bin/varnishd/cache_backend_poll.c)

VCA

Varnish Connection Acceptor -- The code that receives/accepts the TCP connections (bin/varnishd/cache_acceptor.c)

VCC

VCL to C Compiler -- The code that compiles VCL to C code. (lib/libvcl)

VCL

Varnish Configuration Language -- The domain-specific programming language used for configuring a varnishd.

VCT

Varnish CType(3) -- Character classification for RFC2616 and XML parsing.

VDD

Varnish (Core) Developer Day -- Quarterly invite-only meeting strictly for Varnish core (C) developers, packagers and VMOD hackers.

VEV

Varnish EVent -- library functions to implement a simple event-dispatcher.

VGB

Varnish Governing Board -- May or may not exist. If you need to ask, you are not on it.

VGC

Varnish Generated Code -- Code generated by VCC from VCL.

VIN

Varnish Instance Naming -- Resolution of -n arguments.

VLU

Varnish Line Up -- library functions to collect stream of bytes into lines for processing. (lib/libvarnish/vlu.c)

VRE

Varnish Regular-Expression -- library functions for regular-expression based matching and substring replacement. (lib/libvarnish/vre.c)

VRT

Varnish Run Time -- functions called from compiled code. (bin/varnishd/cache_vrt.c)

VRV

VaRY -- Related to processing of Vary: HTTP headers. (bin/varnishd/cache_vary.c)



VSL

Varnish Shared memory Log -- The log written into the shared memory segment for varnish{log,ncsa,top,hist} to see.

VSb

Varnish string Buffer -- a copy of the FreeBSD "sbuf" library, for safe string handling.

VSC

Varnish Statistics Counter -- counters for various stats, exposed via varnishapi.

VSS

Varnish Session Stuff -- library functions to wrap DNS/TCP. (lib/libvarnish/vss.c)

VTC

Varnish Test Code -- a test-specification for the varnishtest program.

VTLA

Varnish Three Letter Acronym -- No rule without an exception.

VUG

Varnish User Group meeting -- Half-yearly event where the users and developers of Varnish Cache gather to share experiences and plan future development.

VWx

Varnish Waiter 'x' -- A code module to monitor idle sessions.

VWE

Varnish Waiter Epoll -- epoll(2) (linux) based waiter module.

VWK

Varnish Waiter Kqueue -- kqueue(2) (freebsd) based waiter module.

VWP

Varnish Waiter Poll -- poll(2) based waiter module.

VWS

Varnish Waiter Solaris -- Solaris ports(2) based waiter module.

