# Varnish Documentation

*Release 3.0.2*

Varnish Project

January 09, 2012

# CONTENTS

**1 Varnish Installation** **3**
1.1 Prerequisites . . . 3
1.2 Installing Varnish . . . 3
1.3 Compiling Varnish from source . . . 4
1.4 Getting hold of us . . . 6
1.5 Reporting bugs . . . 7
1.6 Upgrading from Varnish 2.1 to 3.0 . . . 9

**2 Using Varnish** **13**
2.1 Backend servers . . . 13
2.2 Starting Varnish . . . 14
2.3 Logging in Varnish . . . 14
2.4 Sizing your cache . . . 15
2.5 Put Varnish on port 80 . . . 15
2.6 Varnish Configuration Language - VCL . . . 16
2.7 Statistics . . . 18
2.8 Achieving a high hitrate . . . 19
2.9 Cookies . . . 22
2.10 Vary . . . 23
2.11 Pitfall - Vary: User-Agent . . . 24
2.12 Purging and banning . . . 24
2.13 Edge Side Includes . . . 26
2.14 Running inside a virtual machine (VM) . . . 27
2.15 Advanced Backend configuration . . . 27
2.16 Directors . . . 28
2.17 Health checks . . . 28
2.18 Misbehaving servers . . . 29
2.19 Advanced topics . . . 31
2.20 Troubleshooting Varnish . . . 32

**3 The Varnish Reference Manual** **35**
3.1 VCL . . . 35
3.2 varnish . . . 50
3.3 varnishadm . . . 55
3.4 varnishd . . . 56
3.5 varnishhist . . . 72
3.6 varnishlog . . . 73
3.7 varnishncsa . . . 77
3.8 varnishreplay . . . 79

3.9 varnishsizes . . . . . . . . . . 80
3.10 varnishstat . . . . . . . . . . 81
3.11 varnishtest . . . . . . . . . . 83
3.12 varnishtop . . . . . . . . . . 86
3.13 Shared Memory Logging and Statistics . . . . . . . . . . 88
3.14 VMOD - Varnish Modules . . . . . . . . . . 89
3.15 vmod_std . . . . . . . . . . 92
3.16 Shared Memory Logging . . . . . . . . . . 95

**4 Frequently asked questions** **97**
4.1 General questions . . . . . . . . . . 97
4.2 HTTP . . . . . . . . . . 102
4.3 Configuration . . . . . . . . . . 103
4.4 Logging . . . . . . . . . . 104

**5 Poul-Hennings random outbursts** **105**
5.1 Thoughts on the eve of Varnish 3.0 . . . . . . . . . . 105
5.2 Why no SSL ? . . . . . . . . . . 106
5.3 How GZIP, and GZIP+ESI works in Varnish . . . . . . . . . . 107
5.4 VCL Expressions . . . . . . . . . . 109
5.5 IPv6 Suckage . . . . . . . . . . 110
5.6 What do you mean by 'backend' ? . . . . . . . . . . 111
5.7 Picking platforms . . . . . . . . . . 112
5.8 Security barriers in Varnish . . . . . . . . . . 114
5.9 What were they thinking ? . . . . . . . . . . 115
5.10 Did you call them *autocrap* tools ? . . . . . . . . . . 116
5.11 Why Sphinx and reStructuredText ? . . . . . . . . . . 117

**6 Varnish Glossary** **119**

**7 Indices and tables** **121**

**Index** **123**

Varnish is a state of the art web accelerator. Its mission is to sit in front of a web server and cache content. It makes your web site go faster.

We suggest you start by reading the installation guide *Varnish Installation*. Once you have Varnish up and running go through our tutorial - *Using Varnish*.

Contents:

CHAPTER

ONE

# VARNISH INSTALLATION

This document explains how to get Varnish onto your system, where to get help, how report bugs etc. In other words, it is a manual about pretty much everything else than actually using Varnish to move traffic.

## 1.1 Prerequisites

In order for you to install Varnish you must have the following:

- A fairly modern and 64 bit version of either - Linux - FreeBSD - Solaris
- root access to said system

Varnish can be installed on other UNIX systems as well, but it is not tested particularly well on these plattforms. Varnish is, from time to time, said to work on:

- 32 bit versions of the before-mentioned systems.
- OS X
- NetBSD
- OpenBSD

## 1.2 Installing Varnish

With open source software, you can choose to install binary packages or compile stuff from source-code. To install a package or compile from source is a matter of personal taste. If you don't know which method too choose read the whole document and choose the method you are most comfortable with.

### 1.2.1 Source or packages?

Installing Varnish on most relevant operating systems can usually be done with with the systems package manager, typical examples being:

### 1.2.2 FreeBSD

**From source:** `cd /usr/ports/varnish && make install clean`

**Binary package:** `pkg_add -r varnish`

### 1.2.3 CentOS/RedHat

We try to keep the latest version available as prebuilt RPMs (el5) on *repo.varnish-cache.org <http://repo.varnish-cache.org/>*. See the *RedHat installation instructions <http://www.varnish-cache.org/installation/redhat>* for more information.

Varnish is included in the EPEL repository. Unfortunately we had a syntax change in Varnish 2.0.6->2.1.X. This means that we can not update Varnish in EPEL 5 so the latest version there is Varnish 2.0.6.

EPEL6 should have Varnish 2.1 available once it releases.

### 1.2.4 Debian/Ubuntu

Varnish is distributed with both Debian and Ubuntu. In order to get Varnish up and running type *sudo apt-get install varnish*. Please note that this might not be the latest version of Varnish. If you need a later version of Varnish, please follow the installation instructions for *Debian <http://www.varnish-cache.org/installation/debian>* or *Ubuntu <http://www.varnish-cache.org/installation/ubuntu>*.

### 1.2.5 Other systems

You are probably best of compiling your own code. See Compiling Varnish from source.

If that worked for you, you can skip the rest of this document for now, and and start reading the much more interesting *Using Varnish* instead.

## 1.3 Compiling Varnish from source

If there are no binary packages available for your system, or if you want to compile Varnish from source for other reasons, follow these steps:

We recommend downloading a release tarball, which you can find on *repo.varnish-cache.org <http://repo.varnish-cache.org/source/>*.

Alternatively, if you want to hack on Varnish, you should clone our git repository by doing.

> git clone git://git.varnish-cache.org/varnish-cache

Please note that a git checkout will need some more build-dependencies than listed below, in particular the Python Docutils and Sphinx.

### 1.3.1 Build dependencies on Debian / Ubuntu

In order to build Varnish from source you need a number of packages installed. On a Debian or Ubuntu system these are:

- autotools-dev
- automake1.9
- libtool
- autoconf
- libncurses-dev
- xsltproc

- groff-base
- libpcre3-dev
- pkg-config

### 1.3.2 Build dependencies on Red Hat / CentOS

To build Varnish on a Red Hat or CentOS system you need the following packages installed:

- automake
- autoconf
- libtool
- ncurses-devel
- libxslt
- groff
- pcre-devel
- pkgconfig

### 1.3.3 Configuring and compiling

Next, configuration: The configuration will need the dependencies above satisfied. Once that is taken care of::

```
cd varnish-cache
sh autogen.sh
sh configure
make
```

The `configure` script takes some arguments, but more likely than not, you can forget about that for now, almost everything in Varnish are run time parameters.

Before you install, you may want to run the regression tests, make a cup of tea while it runs, it takes some minutes:

```
make check
```

Don't worry of a single or two tests fail, some of the tests are a bit too timing sensitive (Please tell us which so we can fix it) but if a lot of them fails, and in particular if the `b00000.vtc` test fails, something is horribly wrong, and you will get nowhere without figuring out what.

### 1.3.4 Installing

And finally, the true test of a brave heart:

```
make install
```

Varnish will now be installed in /usr/local. The varnishd binary is in /usr/local/sbin/varnishd and its default configuration will be /usr/local/etc/varnish/default.vcl.

You can now proceed to the *Using Varnish*.

## 1.4 Getting hold of us

Getting hold of the gang behind Varnish is pretty straight forward, we try to help as much as time permits and have tried to streamline this process as much as possible.

But before you grab hold of us, spend a moment composing your thoughts and formulate your question, there is nothing as pointless as simply telling us "Varnish does not work for me" with no further information to give any clue to why.

And before you even do that, do a couple of searches to see if your question is already answered, if it has been, you will get your answer much faster that way.

### 1.4.1 IRC Channel

The most immediate way to get hold of us, is to join our IRC channel:

```
#varnish on server irc.linpro.no
```

The main timezone of the channel is Europe+America.

If you can explain your problem in a few clear sentences, without too much copy&paste, IRC is a good way to try to get help. If you do need to paste log files, VCL and so on, please use a pastebin.

If the channel is all quiet, try again some time later, we do have lives, families and jobs to deal with also.

You are more than welcome to just hang out, and while we don't mind the occational intrusion of the real world into the flow, keep it mostly on topic, and dont paste random links unless they are *really* spectacular and intelligent.

### 1.4.2 Mailing Lists

Getting on or off our mailinglist happens through MailMan.

If you are going to use Varnish, subscribing to our `varnish-announce` mailing list is probably a very good idea. The typical pattern is that people spend some time getting Varnish running, and then more or less forget about it. Therefore the announce list is a good way to be reminded about new releases, bad bugs or security holes.

The `varnish-misc` mailing list is for general banter, questions, suggestions, ideas and so on. If you are new to Varnish it may pay off to subscribe to -misc, simply to have an ear to the telegraph-pole and learn some smart tricks. This is a good place to ask for help with more complex issues, that require quoting of files and long explanations.

Make sure to pick a good subject line, and if the subject of the thread changes, please change the subject to match, some of us deal with hundreds of emails per day, after spam-filters, and we need all the help we can get to pick the interesting ones.

The `varnish-dev` mailing list is used by the developers and is usually quite focused on source-code and such. Everybody on the -dev list is also on -misc, so cross-posting only serves to annoy those people.

### 1.4.3 Trouble Tickets

Please do not open a trouble ticket, unless you have spotted an actual bug in Varnish. Ask on IRC first if you are in doubt.

The reason for this policy, is to avoid the bugs being drowned in a pile of sensible suggestions for future enhancements and call for help from people who forget to check back if they get it and so on.

We track suggestions and ideas in our "Shopping-List" wiki page, and user support via email and IRC.

### 1.4.4 Commercial Support

The following companies offer commercial Varnish support, and are listed here for your convenience. If you want your company listed here, drop an email to phk@FreeBSD.org.

> Varnish Software sales@varnish-software.com

## 1.5 Reporting bugs

Varnish can be a tricky beast to debug, having potentially thousands of threads crowding into a few data structures makes for *interesting* core dumps.

Actually, let me rephrase that without irony: You tire of the "no, not thread 438 either, lets look at 439 then..." routine really fast.

So if you run into a bug, it is important that you spend a little bit of time collecting the right information, to help us fix the bug.

The most valuable information you can give us, is **always** how to trigger and reproduce the problem. If you can tell us that, we rarely need anything else to solve it. The caveat being, that we do not have a way to simulate high levels of real-life web-traffic, so telling us to "have 10.000 clients hit at once" does not really allow us to reproduce.

Roughly we have three clases of bugs with Varnish, and the information we need to debug them depends on the kind of bug.

### 1.5.1 Varnish crashes

Plain and simple: **boom**

Varnish is split over two processes, the manager and the child. The child does all the work, and the manager hangs around to resurect it, if it crashes.

Therefore, the first thing to do if you see a varnish crash, is to examine your syslogs, to see if it has happened before. (One site is rumoured to have had varnish restarting every 10 minutes and *still* provide better service than their CMS system.)

When it crashes, if at all possible, Varnish will spew out a crash dump that looks something like:

```
Child (32619) died signal=6 (core dumped)
Child (32619) Panic message: Assert error in ccf_panic(), cache_cli.c line 153:
  Condition(!strcmp("", "You asked for it")) not true.
errno = 9 (Bad file descriptor)
thread = (cache-main)
ident = FreeBSD,9.0-CURRENT,amd64,-sfile,-hcritbit,kqueue
Backtrace:
  0x42bce1: pan_ic+171
  0x4196af: ccf_panic+4f
  0x8006b3ef2: _end+80013339a
  0x8006b4307: _end+8001337af
  0x8006b8b76: _end+80013801e
  0x8006b8d84: _end+80013822c
  0x8006b51c1: _end+800134669
  0x4193f6: CLI_Run+86
  0x429f8b: child_main+14b
  0x43ef68: start_child+3f8
[...]
```

If you can get that information to us, we are usually able to see exactly where things went haywire, and that speeds up bugfixing a lot.

There will be a lot more information than this, and before sending it all to us, you should obscure any sensitive/secret data/cookies/passwords/ip# etc. Please make sure to keep context when you do so, ie: do not change all the IP# to "X.X.X.X", but change each IP# to something unique, otherwise we are likely to be more confused than informed.

The most important line is the "Panic Message", which comes in two general forms:

**"Missing errorhandling code in ..."** This is a place where we can conceive ending up, and have not (yet) written the padded-box error handling code for.

The most likely cause here, is that you need a larger workspace for HTTP headers and Cookies. (XXX: which params to tweak)

Please try that before reporting a bug.

**"Assert error in ..."** This is something bad that should never happen, and a bug report is almost certainly in order. As always, if in doubt ask us on IRC before opening the ticket.

In your syslog it may all be joined into one single line, but if you can reproduce the crash, do so while running varnishd manually:

```
varnishd -d <your other arguments> |& tee /tmp/_catch_bug
```

That will get you the entire panic message into a file.

(Remember to type `start` to launch the worker process, that is not automatic when `-d` is used.)

## 1.5.2 Varnish goes on vacation

This kind of bug is nasty to debug, because usually people tend to kill the process and send us an email saying "Varnish hung, I restarted it" which gives us only about 1.01 bit of usable debug information to work with.

What we need here is all the information can you squeeze out of your operating system **before** you kill the Varnish process.

One of the most valuable bits of information, is if all Varnish' threads are waiting for something or if one of them is spinning furiously on some futile condition.

Commands like `top -H` or `ps -Haxlw` or `ps -efH` should be able to figure that out.

If one or more threads are spinning, use `strace` or `ktrace` or `truss` (or whatever else your OS provides) to get a trace of which system calls the varnish process issues. Be aware that this may generate a lot of very repetitive data, usually one second worth is more than enough.

Also, run `varnishlog` for a second, and collect the output for us, and if `varnishstat` shows any activity, capture that also.

When you have done this, kill the Varnish *child* process, and let the *master* process restart it. Remember to tell us if that does or does not work. If it does not, kill all Varnish processes, and start from scratch. If that does not work either, tell us, that means that we have wedged your kernel.

## 1.5.3 Varnish does something wrong

These are the easy bugs: usually all we need from you is the relevant transactions recorded with `varnishlog` and your explanation of what is wrong about what Varnish does.

Be aware, that often Varnish does exactly what you asked it to, rather than what you intended it to do, so it sounds like a bug that would have tripped up everybody else, take a moment to read though your VCL and see if it really does what you think.

You can also try setting the `vcl_trace` parameter, that will generate log records with like and char number for each statement executed in your VCL program.

## 1.6 Upgrading from Varnish 2.1 to 3.0

This is a compilation of items you need to pay attention to when upgrading from Varnish 2.1 to 3.0

### 1.6.1 Changes to VCL

In most cases you need to update your VCL since there has been some changes to the syntax.

#### string concatenation operator

String concatenation did not have an operator previously, but this has now been changed to `+`.

#### no more %-escapes in strings

To simplify strings, the %-encoding has been removed. If you need non-printable characters, you need to use inline C.

#### `log` moved to the std vmod

`log` has moved to the std vmod:

```
log "log something";
```

becomes:

```
import std;
std.log("log something");
```

You only need to import std once.

#### purges are now called bans

`purge()` and `purge_url()` are now respectively `ban()` and `ban_url()`, so you should replace all occurences:

```
purge("req.url = " req.url);
```

becomes:

```
ban("req.url = " + req.url);
```

`purge` does not take any arguments anymore, but can be used in vcl_hit or vcl_miss to purge the item from the cache, where you would reduce ttl to 0 in Varnish 2.1:

```
sub vcl_hit {
  if (req.request == "PURGE") {
    set obj.ttl = 0s;
    error 200 "Purged.";
  }
}
```

becomes:

```
sub vcl_hit {
  if (req.request == "PURGE") {
    purge;
    error 200 "Purged.";
  }
}
```

### `beresp.cacheable` is gone

`beresp.cacheable` is gone, and can be replaced with `beresp.ttl > 0s`

### returns are now done with the `return()` function

`pass`, `pipe`, `lookup`, `deliver`, `fetch`, `hash`, `pipe` and `restart` are no longer keywords, but arguments to `return()`, so:

```
sub vcl_pass {
  pass;
}
```

becomes:

```
sub vcl_pass {
  return(pass);
}
```

### `req.hash` is replaced with `hash_data()`

You no longer append to the hash with +=, so:

```
set req.hash += req.url;
```

becomes:

```
hash_data(req.url);
```

### `esi` is replaced with `beresp.do_esi`

You no longer enable ESI with `esi`, so:

```
esi;
```

in `vcl_fetch` becomes:

```
set beresp.do_esi = true;
```

### `pass` in `vcl_fetch` renamed to `hit_for_pass`

The difference in behaviour of `pass` in `vcl_recv` and `vcl_fetch` confused people, so to make it clearer that they are different, you must now do `return(hit_for_pass)` when doing a pass in `vcl_fetch`.

### 1.6.2 Changes to behaviour

Varnish will return an error when headers are too large instead of just ignoring them. If the limits are too low, Varnish will return HTTP 413. You can change the limits by increasing http_req_hdr_len and http_req_size.

thread_pool_max is now per thread pool, while it was a total across all pools in 2.1. If you had this set in 2.1, you should adjust it for 3.0.

CHAPTER

TWO

# USING VARNISH

This tutorial is intended for system administrators managing Varnish cache. The reader should know how to configure her web- or application server and have basic knowledge of the HTTP protocol. The reader should have Varnish up and running with the default configuration.

The tutorial is split into short chapters, each chapter taking on a separate topic. Good luck.

## 2.1 Backend servers

Varnish has a concept of “backend” or “origin” servers. A backend server is the server providing the content Varnish will accelerate.

Our first task is to tell Varnish where it can find its content. Start your favorite text editor and open the varnish default configuration file. If you installed from source this is /usr/local/etc/varnish/default.vcl, if you installed from a package it is probably /etc/varnish/default.vcl.

Somewhere in the top there will be a section that looks a bit like this.:

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

We comment in this bit of text and change the port setting from 8080 to 80, making the text look like.:

```
backend default {
      .host = "127.0.0.1";
      .port = "80";
}
```

Now, this piece of configuration defines a backend in Varnish called *default*. When Varnish needs to get content from this backend it will connect to port 80 on localhost (127.0.0.1).

Varnish can have several backends defined and can you can even join several backends together into clusters of backends for load balancing purposes.

Now that we have the basic Varnish configuration done, let us start up Varnish on port 8080 so we can do some fundamental testing on it.

## 2.2 Starting Varnish

I assume varnishd is in your path. You might want to run `pkill varnishd` to make sure varnishd isn't running. Become root and type:

```
# varnishd -f /usr/local/etc/varnish/default.vcl -s malloc,1G -T
127.0.0.1:2000 -a 0.0.0.0:8080
```

I added a few options, lets go through them:

**`-f /usr/local/etc/varnish/default.vcl`** The -f options specifies what configuration varnishd should use.

**`-s malloc,1G`** The -s options chooses the storage type Varnish should use for storing its content. I used the type *malloc*, which just uses memory for storage. There are other backends as well, described in :ref:tutorial-storage. 1G specifies how much memory should be allocated - one gigabyte.

**`-T 127.0.0.1:2000`** Varnish has a built-in text-based administration interface. Activating the interface makes Varnish manageble without stopping it. You can specify what interface the management interface should listen to. Make sure you don't expose the management interface to the world as you can easily gain root access to a system via the Varnish management interface. I recommend tieing it to localhost. If you have users on your system that you don't fully trust, use firewall rules to restrict access to the interface to root only.

**`-a 0.0.0.0:8080`** I specify that I want Varnish to listen on port 8080 for incomming HTTP requests. For a production environment you would probably make Varnish listen on port 80, which is the default.

Now you have Varnish running. Let us make sure that it works properly. Use your browser to go to http://192.168.2.2:8080/ (obviously, you should replace the IP address with one on your own system) - you should now see your web application running there.

Whether or not the application actually goes faster when run through Varnish depends on a few factors. If you application uses cookies for every session (a lot of PHP and Java applications seem to send a session cookie if it is needed or not) or if it uses authentication chances are Varnish won't do much caching. Ignore that for the moment, we come back to that in *Achieving a high hitrate*.

Lets make sure that Varnish really does do something to your web site. To do that we'll take a look at the logs.

## 2.3 Logging in Varnish

One of the really nice features in Varnish is how logging works. Instead of logging to normal log file Varnish logs to a shared memory segment. When the end of the segment is reached we start over, overwriting old data. This is much, much faster then logging to a file and it doesn't require disk space.

The flip side is that is you forget to have program actually write the logs to disk they will disappear.

varnishlog is one of the programs you can use to look at what Varnish is logging. Varnishlog gives you the raw logs, everything that is written to the logs. There are other clients as well, we'll show you these later.

In the terminal window you started varnish now type *varnishlog* and press enter.

You'll see lines like these scrolling slowly by.:

```
0 CLI          - Rd ping
0 CLI          - Wr 200 PONG 1273698726 1.0
```

These is the Varnish master process checking up on the caching process to see that everything is OK.

Now go to the browser and reload the page displaying your web app. You'll see lines like these.:

```
11 SessionOpen  c 127.0.0.1 58912 0.0.0.0:8080
11 ReqStart     c 127.0.0.1 58912 595005213
11 RxRequest    c GET
11 RxURL        c /
11 RxProtocol   c HTTP/1.1
11 RxHeader     c Host: localhost:8080
11 RxHeader     c Connection: keep-alive
```

The first column is an arbitrary number, it defines the request. Lines with the same number are part of the same HTTP transaction. The second column is the *tag* of the log message. All log entries are tagged with a tag indicating what sort of activity is being logged. Tags starting with Rx indicate Varnish is recieving data and Tx indicates sending data.

The third column tell us whether this is is data coming or going to the client (c) or to/from the backend (b). The forth column is the data being logged.

Now, you can filter quite a bit with varnishlog. The basic option you want to know are:

| | |
|---|---|
| **-b** | Only show log lines from traffic going between Varnish and the backend servers. This will be useful when we want to optimize cache hit rates. |
| **-c** | Same as -b but for client side traffic. |
| **-i tag** | Only show lines with a certain tag. "varnishlog -i SessionOpen" will only give you new sessions. Note that the tags are case insensitive. |
| **-I Regex** | Filter the data through a regex and only show the matching lines. To show all cookie headers coming from the clients: `$ varnishlog -c -i RxHeader -I Cookie` |
| **-o** | Group log entries by request ID. |

Now that Varnish seem to work OK it's time to put Varnish on port 80 while we tune it.

## 2.4 Sizing your cache

Picking how much memory you should give Varnish can be a tricky task. A few things to consider:

- How big is your *hot* data set. For a portal or news site that would be the size of the front page with all the stuff on it, and the size of all the pages and objects linked from the first page.
- How expensive is it to generate an object? Sometimes it makes sense to only cache images a little while or not to cache them at all if they are cheap to serve from the backend and you have a limited amount of memory.
- Watch the n_lru_nuked counter with *varnishstat* or some other tool. If you have a lot of LRU activity then your cache is evicting objects due to space constraints and you should consider increasing the size of the cache.

Be aware that every object that is stored also carries overhead that is kept outside the actually storage area. So, even if you specify -s malloc,16G varnish might actually use **double** that. Varnish has a overhead of about 1k per object. So, if you have lots of small objects in your cache the overhead might be significant.

## 2.5 Put Varnish on port 80

Until now we've been running with Varnish on a high port, for testing purposes. You should test your application and if it works OK we can switch, so Varnish will be running on port 80 and your web server on a high port.

First we kill off varnishd:

```
# pkill varnishd
```

and stop your web server. Edit the configuration for your web server and make it bind to port 8080 instead of 80. Now open the Varnish default.vcl and change the port of the *default* backend to 8080.

Start up your web server and then start varnish:

```
# varnishd -f /usr/local/etc/varnish/default.vcl -s malloc,1G -T 127.0.0.1:2000
```

Note that we've removed the -a option. Now Varnish, as its default setting dictates, will bind to the http port (80). Now everyone accessing your site will be accessing through Varnish.

## 2.6 Varnish Configuration Language - VCL

Varnish has a great configuration system. Most other systems use configuration directives, where you basically turn on and off lots of switches. Varnish uses a domain specific language called Varnish Configuration Language, or VCL for short. Varnish translates this configuration into binary code which is then executed when requests arrive.

The VCL files are divided into subroutines. The different subroutines are executed at different times. One is executed when we get the request, another when files are fetched from the backend server.

Varnish will execute these subroutines of code at different stages of its work. Because it is code it is execute line by line precedence isn't a problem. At some point you call an action in this subroutine and then the execution of the subroutine stops.

If you don't call an action in your subroutine and it reaches the end Varnish will execute some built in VCL code. You will see this VCL code commented out in default.vcl.

99% of all the changes you'll need to do will be done in two of these subroutines. *vcl_recv* and *vcl_fetch*.

### 2.6.1 vcl_recv

vcl_recv (yes, we're skimpy with characters, it's Unix) is called at the beginning of a request, after the complete request has been received and parsed. Its purpose is to decide whether or not to serve the request, how to do it, and, if applicable, which backend to use.

In vcl_recv you can also alter the request. Typically you can alter the cookies and add and remove request headers.

Note that in vcl_recv only the request object, req is available.

### 2.6.2 vcl_fetch

vcl_fetch is called *after* a document has been successfully retrieved from the backend. Normal tasks her are to alter the response headers, trigger ESI processing, try alternate backend servers in case the request failed.

In vcl_fetch you still have the request object, req, available. There is also a *backend response*, beresp. beresp will contain the HTTP headers from the backend.

### 2.6.3 actions

The most common actions to return are these:

***pass*** When you return pass the request and subsequent response will be passed to and from the backend server. It won't be cached. pass can be returned from vcl_recv

***hit_for_pass*** Similar to pass, but accessible from vcl_fetch. Unlike pass, hit_for_pass will create a hitforpass object in the cache. This has the side-effect of caching the decision not to cache. This is to allow would-be uncachable requests to be passed to the backend at the same time. The same logic is not necessary in vcl_recv because this happens before any potential queueing for an object takes place.

***lookup*** When you return lookup from vcl_recv you tell Varnish to deliver content from cache even if the request othervise indicates that the request should be passed. You can't return lookup from vcl_fetch.

***pipe*** Pipe can be returned from vcl_recv as well. Pipe short circuits the client and the backend connections and Varnish will just sit there and shuffle bytes back and forth. Varnish will not look at the data being send back and forth - so your logs will be incomplete. Beware that with HTTP 1.1 a client can send several requests on the same connection and so you should instruct Varnish to add a "Connection: close" header before actually returning pipe.

***deliver*** Deliver the cached object to the client. Usually returned from vcl_fetch.

## 2.6.4 Requests, responses and objects

In VCL, there are three important data structures. The request, coming from the client, the response coming from the backend server and the object, stored in cache.

In VCL you should know the following structures.

***req*** The request object. When Varnish has received the request the req object is created and populated. Most of the work you do in vcl_recv you do on or with the req object.

***beresp*** The backend respons object. It contains the headers of the object comming from the backend. Most of the work you do in vcl_fetch you do on the beresp object.

***obj*** The cached object. Mostly a read only object that resides in memory. obj.ttl is writable, the rest is read only.

## 2.6.5 Operators

The following operators are available in VCL. See the examples further down for, uhm, examples.

**=** Assignment operator.

**==** Comparison.

**~** Match. Can either be used with regular expressions or ACLs.

**!** Negation.

**&&** Logical *and*

**||** Logical *or*

## 2.6.6 Example 1 - manipulating headers

Lets say we want to remove the cookie for all objects in the /static directory of our web server::

```
sub vcl_recv {
  if (req.url ~ "^/images") {
    unset req.http.cookie;
  }
}
```

Now, when the request is handled to the backend server there will be no cookie header. The interesting line is the one with the if-statement. It matches the URL, taken from the request object, and matches it against the regular expression. Note the match operator. If it matches the Cookie: header of the request is unset (deleted).

### 2.6.7 Example 2 - manipulating beresp

Here we override the TTL of a object comming from the backend if it matches certain criteria::

```
sub vcl_fetch {
   if (req.url ~ "\.(png|gif|jpg)$") {
     unset beresp.http.set-cookie;
     set beresp.ttl = 1h;
  }
}
```

### 2.6.8 Example 3 - ACLs

You create a named access control list with the *acl* keyword. You can match the IP address of the client against an ACL with the match operator.:

```
# Who is allowed to purge....
acl local {
    "localhost";
    "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.request == "PURGE") {
    if (client.ip ~ local) {
       return(lookup);
    }
  }
}

sub vcl_hit {
   if (req.request == "PURGE") {
     set obj.ttl = 0s;
     error 200 "Purged.";
    }
}

sub vcl_miss {
  if (req.request == "PURGE") {
    error 404 "Not in cache.";
  }
}
```

## 2.7 Statistics

Now that your varnish is up and running let's have a look at how it is doing. There are several tools that can help.

### 2.7.1 varnishtop

The varnishtop utility reads the shared memory logs and presents a continuously updated list of the most commonly occurring log entries.

With suitable filtering using the -I, -i, -X and -x options, it can be used to display a ranking of requested documents, clients, user agents, or any other information which is recorded in the log.

`varnishtop -i rxurl` will show you what URLs are being asked for by the client. `varnishtop -i txurl` will show you what your backend is being asked the most. `varnishtop -i RxHeader -I Accept-Encoding` will show the most popular Accept-Encoding header the client are sending you.

### 2.7.2 varnishhist

The varnishhist utility reads varnishd(1) shared memory logs and presents a continuously updated histogram showing the distribution of the last N requests by their processing. The value of N and the vertical scale are displayed in the top left corner. The horizontal scale is logarithmic. Hits are marked with a pipe character ("|"), and misses are marked with a hash character ("#").

### 2.7.3 varnishsizes

Varnishsizes does the same as varnishhist, except it shows the size of the objects and not the time take to complete the request. This gives you a good overview of how big the objects you are serving are.

### 2.7.4 varnishstat

Varnish has lots of counters. We count misses, hits, information about the storage, threads created, deleted objects. Just about everything. varnishstat will dump these counters. This is useful when tuning varnish.

There are programs that can poll varnishstat regularly and make nice graphs of these counters. One such program is Munin. Munin can be found at http://munin-monitoring.org/ . There is a plugin for munin in the varnish source code.

## 2.8 Achieving a high hitrate

Now that Varnish is up and running, and you can access your web application through Varnish. Unless your application is specifically written to work behind a web accelerator you'll probably need to do some changes to either the configuration or the application in order to get a high hit rate in Varnish.

Varnish will not cache your data unless it's absolutely sure it is safe to do so. So, for you to understand how Varnish decides if and how to cache a page, I'll guide you through a couple of tools that you will find useful.

Note that you need a tool to see what HTTP headers fly between you and the web server. On the Varnish server, the easiest is to use varnishlog and varnishtop but sometimes a client-side tool makes sense. Here are the ones I use.

### 2.8.1 Tool: varnishtop

You can use varnishtop to identify what URLs are hitting the backend the most. `varnishtop -i txurl` is an essential command. You can see some other examples of varnishtop usage in *Statistics*.

### 2.8.2 Tool: varnishlog

When you have identified the an URL which is frequently sent to the backend you can use varnishlog to have a look at the whole request. `varnishlog -c -o /foo/bar` will give the whole (-o) requests coming from the client (-c) matching /foo/bar.

For extended diagnostics headers, see http://www.varnish-cache.org/trac/wiki/VCLExampleHitMissHeader

### 2.8.3 Tool: lwp-request

lwp-request is part of The World-Wide Web library for Perl. It's a couple of really basic programs that can execute an HTTP request and give you the result. I mostly use two programs, GET and HEAD.

vg.no was the first site to use Varnish and the people running Varnish there are quite clueful. So it's interesting to look at their HTTP Headers. Let's send a GET request for their home page:

```
$ GET -H 'Host: www.vg.no' -Used http://vg.no/
GET http://vg.no/
Host: www.vg.no
User-Agent: lwp-request/5.834 libwww-perl/5.834

200 OK
Cache-Control: must-revalidate
Refresh: 600
Title: VG Nett - Forsiden - VG Nett
X-Age: 463
X-Cache: HIT
X-Rick-Would-Never: Let you down
X-VG-Jobb: http://www.finn.no/finn/job/fulltime/result?keyword=vg+multimedia Merk:HeaderNinja
X-VG-Korken: http://www.youtube.com/watch?v=Fcj8CnD5188
X-VG-WebCache: joanie
X-VG-WebServer: leon
```

OK. Let me explain what it does. GET usually sends off HTTP 0.9 requests, which lack the Host header. So I add a Host header with the -H option. -U print request headers, -s prints response status, -e prints response headers and -d discards the actual content. We don't really care about the content, only the headers.

As you can see, VG adds quite a bit of information in their headers. Some of the headers, like the X-Rick-Would-Never are specific to vg.no and their somewhat odd sense of humour. Others, like the X-VG-Webcache are for debugging purposes.

So, to check whether a site sets cookies for a specific URL, just do:

```
GET -Used http://example.com/ |grep ^Set-Cookie
```

### 2.8.4 Tool: Live HTTP Headers

There is also a plugin for Firefox. *Live HTTP Headers* can show you what headers are being sent and recieved. Live HTTP Headers can be found at https://addons.mozilla.org/en-US/firefox/addon/3829/ or by googling "Live HTTP Headers".

### 2.8.5 The role of HTTP Headers

Along with each HTTP request and reponse comes a bunch of headers carrying metadata. Varnish will look at these headers to determine if it is appropriate to cache the contents and how long Varnish can keep the content.

Please note that when considering these headers Varnish actually considers itself *part of* the actual webserver. The rationale being that both are under your control.

The term *surrogate origin cache* is not really well defined by the IETF so RFC 2616 so the various ways Varnish works might differ from your expectations.

Let's take a look at the important headers you should be aware of:

### 2.8.6 Cache-Control

The Cache-Control instructs caches how to handle the content. Varnish cares about the *max-age* parameter and uses it to calculate the TTL for an object.

"Cache-Control: nocache" is ignored but if you need this you can easily add support for it.

So make sure you issue a Cache-Control header with a max-age header. You can have a look at what Varnish Software's drupal server issues:

```
$ GET -Used http://www.varnish-software.com/|grep ^Cache-Control
Cache-Control: public, max-age=600
```

### 2.8.7 Age

Varnish adds an Age header to indicate how long the object has been kept inside Varnish. You can grep out Age from varnishlog like this:

```
varnishlog -i TxHeader -I ^Age
```

### 2.8.8 Pragma

An HTTP 1.0 server might send "Pragma: nocache". Varnish ignores this header. You could easily add support for this header in VCL.

In vcl_fetch:

```
if (beresp.http.Pragma ~ "nocache") {
   pass;
}
```

### 2.8.9 Authorization

If Varnish sees an Authorization header it will pass the request. If this is not what you want you can unset the header.

### 2.8.10 Overriding the time-to-live (ttl)

Sometimes your backend will misbehave. It might, depending on your setup, be easier to override the ttl in Varnish than to fix your somewhat cumbersome backend.

You need VCL to identify the objects you want and then you set the beresp.ttl to whatever you want:

```
sub vcl_fetch {
    if (req.url ~ "^/legacy_broken_cms/") {
        set beresp.ttl = 5d;
    }
}
```

The example will set the TTL to 5 days for the old legacy stuff on your site.

### 2.8.11 Forcing caching for certain requests and certain responses

Since you still have this cumbersome backend that isn't very friendly to work with you might want to override more stuff in Varnish. We recommend that you rely as much as you can on the default caching rules. It is perfectly easy to force Varnish to lookup an object in the cache but it isn't really recommended.

### 2.8.12 Normalizing your namespace

Some sites are accessed via lots of hostnames. http://www.varnish-software.com/, http://varnish-software.com/ and http://varnishsoftware.com/ all point at the same site. Since Varnish doesn't know they are different, Varnish will cache different versions of every page for every hostname. You can mitigate this in your web server configuration by setting up redirects or by using the following VCL:

```
if (req.http.host ~ "(?i)^(www.)?varnish-?software.com") {
  set req.http.host = "varnish-software.com";
}
```

### 2.8.13 Ways of increasing your hitrate even more

The following chapters should give your ways of further increasing your hitrate, especially the chapter on Cookies.

- *Cookies*
- *Vary*
- *Purging and banning*
- *Edge Side Includes*

## 2.9 Cookies

Varnish will not cache a object coming from the backend with a Set-Cookie header present. Also, if the client sends a Cookie header, Varnish will bypass the cache and go directly to the backend.

This can be overly conservative. A lot of sites use Google Analytics (GA) to analyze their traffic. GA sets a cookie to track you. This cookie is used by the client side java script and is therefore of no interest to the server.

For a lot of web application it makes sense to completely disregard the cookies unless you are accessing a special part of the web site. This VCL snippet in vcl_recv will disregard cookies unless you are accessing /admin/:

```
if ( !( req.url ~ ^/admin/) ) {
  unset req.http.Cookie;
}
```

Quite simple. If, however, you need to do something more complicated, like removing one out of several cookies, things get difficult. Unfortunately Varnish doesn't have good tools for manipulating the Cookies. We have to use regular expressions to do the work. If you are familiar with regular expressions you'll understand whats going on. If you don't I suggest you either pick up a book on the subject, read through the *pcrepattern* man page or read through one of many online guides.

Let me show you what Varnish Software uses. We use some cookies for Google Analytics tracking and similar tools. The cookies are all set and used by Javascript. Varnish and Drupal doesn't need to see those cookies and since Varnish will cease caching of pages when the client sends cookies we will discard these unnecessary cookies in VCL.

In the following VCL we discard all cookies that start with a underscore:

```
// Remove has_js and Google Analytics __* cookies.
set req.http.Cookie = regsuball(req.http.Cookie, "(^|;\s*)(_[_a-z]+|has_js)=[^;]*", "");
// Remove a ";" prefix, if present.
set req.http.Cookie = regsub(req.http.Cookie, "^;\s*", "");
```

Let me show you an example where we remove everything except the cookies named COOKIE1 and COOKIE2 and you can marvel at it:

```
sub vcl_recv {
  if (req.http.Cookie) {
    set req.http.Cookie = ";" req.http.Cookie;
    set req.http.Cookie = regsuball(req.http.Cookie, "; +", ";");
    set req.http.Cookie = regsuball(req.http.Cookie, ";(COOKIE1|COOKIE2)=", "; \1=");
    set req.http.Cookie = regsuball(req.http.Cookie, ";[^ ][^;]*", "");
    set req.http.Cookie = regsuball(req.http.Cookie, "^[; ]+|[; ]+$", "");

    if (req.http.Cookie == "") {
        remove req.http.Cookie;
    }
}
```

The example is taken from the Varnish Wiki, where you can find other scary examples of what can be done i VCL.

## 2.10 Vary

The Vary header is sent by the web server to indicate what makes a HTTP object Vary. This makes a lot of sense with headers like Accept-Encoding. When a server issues a "Vary: Accept-Encoding" it tells Varnish that its needs to cache a separate version for every different Accept-Encoding that is coming from the clients. So, if a clients only accepts gzip encoding Varnish won't serve the version of the page encoded with the deflate encoding.

The problem is that the Accept-Encoding field contains a lot of different encodings. If one browser sends:

```
Accept-Encodign: gzip,deflate
```

And another one sends:

```
Accept-Encoding:: deflate,gzip
```

Varnish will keep two variants of the page requested due to the different Accept-Encoding headers. Normalizing the accept-encoding header will sure that you have as few variants as possible. The following VCL code will normalize the Accept-Encoding headers.:

```
if (req.http.Accept-Encoding) {
    if (req.url ~ "\.(jpg|png|gif|gz|tgz|bz2|tbz|mp3|ogg)$") {
        # No point in compressing these
        remove req.http.Accept-Encoding;
    } elsif (req.http.Accept-Encoding ~ "gzip") {
        set req.http.Accept-Encoding = "gzip";
    } elsif (req.http.Accept-Encoding ~ "deflate") {
        set req.http.Accept-Encoding = "deflate";
    } else {
        # unkown algorithm
        remove req.http.Accept-Encoding;
    }
}
```

The code sets the Accept-Encoding header from the client to either gzip, deflate with a preference for gzip.

## 2.11 Pitfall - Vary: User-Agent

Some applications or application servers send *Vary: User-Agent* along with their content. This instructs Varnish to cache a separate copy for every variation of User-Agent there is. There are plenty. Even a single patchlevel of the same browser will generate at least 10 different User-Agent headers based just on what operating system they are running.

So if you *really* need to Vary based on User-Agent be sure to normalize the header or your hit rate will suffer badly. Use the above code as a template.

## 2.12 Purging and banning

One of the most effective way of increasing your hit ratio is to increase the time-to-live (ttl) of your objects. But, as you're aware of, in this twitterific day of age serving content that is outdated is bad for business.

The solution is to notify Varnish when there is fresh content available. This can be done through two mechanisms. HTTP purging and bans. First, let me explain the HTTP purges.

### 2.12.1 HTTP Purges

A *purge* is what happens when you pick out an object from the cache and discard it along with its variants. Usually a purge is invoked through HTTP with the method PURGE.

An HTTP purge is similar to an HTTP GET request, except that the *method* is PURGE. Actually you can call the method whatever you'd like, but most people refer to this as purging. Squid supports the same mechanism. In order to support purging in Varnish you need the following VCL in place:

```
acl purge {
        "localhost";
        "192.168.55.0/24";
}

sub vcl_recv {
        # allow PURGE from localhost and 192.168.55...

        if (req.request == "PURGE") {
                if (!client.ip ~ purge) {
                        error 405 "Not allowed.";
                }
                return (lookup);
        }
}
```

```
sub vcl_hit {
        if (req.request == "PURGE") {
                purge;
                error 200 "Purged.";
        }
}

sub vcl_miss {
        if (req.request == "PURGE") {
                purge;
                error 200 "Purged.";
        }
}
```

As you can see we have used to new VCL subroutines, vcl_hit and vcl_miss. When we call lookup Varnish will try to lookup the object in its cache. It will either hit an object or miss it and so the corresponding subroutine is called. In vcl_hit the object that is stored in cache is available and we can set the TTL.

So for example.com to invalidate their front page they would call out to Varnish like this:

```
PURGE / HTTP/1.0
Host: example.com
```

And Varnish would then discard the front page. This will remove all variants as defined by Vary.

### 2.12.2 Bans

There is another way to invalidate content. Bans. You can think of bans as a sort of a filter. You *ban* certain content from being served from your cache. You can ban content based on any metadata we have.

Support for bans is built into Varnish and available in the CLI interface. For VG to ban every png object belonging on example.com they could issue:

```
ban req.http.host == "example.com" && req.http.url ~ "\.png$"
```

Quite powerful, really.

Bans are checked when we hit an object in the cache, but before we deliver it. *An object is only checked against newer bans.*

Bans that only match against beresp.* are also processed by a background worker threads called the *ban lurker*. The ban lurker will walk the heap and try to match objects and will evict the matching objects. How aggressive the ban lurker is can be controlled by the parameter ban_lurker_sleep.

Bans that are older then the oldest objects in the cache are discarded without evaluation. If you have a lot of objects with long TTL, that are seldom accessed you might accumulate a lot of bans. This might impact CPU usage and thereby performance.

You can also add bans to Varnish via HTTP. Doing so requires a bit of VCL:

```
sub vcl_recv {
        if (req.request == "BAN") {
                # Same ACL check as above:
                if (!client.ip ~ purge) {
                        error 405 "Not allowed.";
                }
                ban("req.http.host == " + req.http.host +
                      "&& req.url == " + req.url);
                # Throw a synthetic page so the
                # request won’t go to the backend.
                error 200 "Ban added";
        }
}
```

This VCL sniplet enables Varnish to handle an HTTP BAN method, adding a ban on the URL, including the host part.

## 2.13 Edge Side Includes

*Edge Side Includes* is a language to include *fragments* of web pages in other web pages. Think of it as HTML include statement that works over HTTP.

On most web sites a lot of content is shared between pages. Regenerating this content for every page view is wasteful and ESI tries to address that letting you decide the cache policy for each fragment individually.

In Varnish we’ve only implemented a small subset of ESI. As of 2.1 we have three ESI statements:

- esi:include
- esi:remove
- <!–esi ...–>

Content substitution based on variables and cookies is not implemented but is on the roadmap.

### 2.13.1 Example: esi include

Lets see an example how this could be used. This simple cgi script outputs the date::

```
#!/bin/sh

echo 'Content-type: text/html'
echo ''
date "+%Y-%m-%d %H:%M"
```

Now, lets have an HTML file that has an ESI include statement::

```
<HTML>
<BODY>
The time is: <esi:include src="/cgi-bin/date.cgi"/>
at this very moment.
</BODY>
</HTML>
```

For ESI to work you need to activate ESI processing in VCL, like this::

```
sub vcl_fetch {
    if (req.url == "/test.html") {
       set beresp.do_esi = true; /* Do ESI processing               */
       set beresp.ttl = 24 h;    /* Sets the TTL on the HTML above  */
    } elseif (req.url == "/cgi-bin/date.cgi") {
       set beresp.ttl = 1m;      /* Sets a one minute TTL on        */
                                 /*  the included object            */
    }
}
```

### 2.13.2 Example: esi remove

The *remove* keyword allows you to remove output. You can use this to make a fall back of sorts, when ESI is not available, like this::

```
<esi:include src="http://www.example.com/ad.html"/>
<esi:remove>
  <a href="http://www.example.com">www.example.com</a>
</esi:remove>
```

### 2.13.3 Example: <!–esi ... –>

This is a special construct to allow HTML marked up with ESI to render without processing. ESI Processors will remove the start ("<!–esi") and end ("–>") when the page is processed, while still processing the contents. If the page is not processed, it will remain, becoming an HTML/XML comment tag. For example:

```
<!--esi
<p>Warning: ESI Disabled!</p>
</p>  -->
```

This assures that the ESI markup will not interfere with the rendering of the final HTML if not processed.

## 2.14 Running inside a virtual machine (VM)

It is possible, but not recommended for high performance, to run Varnish on virtualised hardware.

### 2.14.1 OpenVZ

If you are running on 64bit OpenVZ (or Parallels VPS), you must reduce the maximum stack size before starting Varnish. The default allocates to much memory per thread, which will make varnish fail as soon as the number of threads (==traffic) increases.

Reduce the maximum stack size by running:

```
ulimit -s 256
```

in the startup script.

## 2.15 Advanced Backend configuration

At some point you might need Varnish to cache content from several servers. You might want Varnish to map all the URL into one single host or not. There are lot of options.

Lets say we need to introduce a Java application into out PHP web site. Lets say our Java application should handle URL beginning with /java/.

We manage to get the thing up and running on port 8000. Now, lets have a look a default.vcl.:

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

We add a new backend.:

```
backend java {
    .host = "127.0.0.1";
    .port = "8000";
}
```

Now we need tell where to send the difference URL. Lets look at vcl_recv.:

```
sub vcl_recv {
    if (req.url ~ "^/java/") {
        set req.backend = java;
    } else {
        set req.backend = default.
    }
}
```

It's quite simple, really. Lets stop and think about this for a moment. As you can see you can define how you choose backends based on really arbitrary data. You want to send mobile devices to a different backend? No problem. if (req.User-agent ~ /mobile/) .... should do the trick.

## 2.16 Directors

You can also group several backend into a group of backends. These groups are called directors. This will give you increased performance and resilience. You can define several backends and group them together in a director.:

```
backend server1 {
    .host = "192.168.0.10";
}
backend server2{
    .host = "192.168.0.10";
}
```

Now we create the director.:

```
director example_director round-robin {
{
        .backend = server1;
}
# server2
{
        .backend = server2;
}
# foo
}
```

This director is a round-robin director. This means the director will distribute the incoming requests on a round-robin basis. There is also a *random* director which distributes requests in a, you guessed it, random fashion.

But what if one of your servers goes down? Can Varnish direct all the requests to the healthy server? Sure it can. This is where the Health Checks come into play.

## 2.17 Health checks

Lets set up a director with two backends and health checks. First lets define the backends.:

```
backend server1 {
  .host = "server1.example.com";
  .probe = {
         .url = "/";
         .interval = 5s;
         .timeout = 1 s;
         .window = 5;
         .threshold = 3;
    }
  }
backend server2 {
   .host = "server2.example.com";
   .probe = {
         .url = "/";
         .interval = 5s;
         .timeout = 1 s;
         .window = 5;
         .threshold = 3;
   }
 }
```

Whats new here is the probe. Varnish will check the health of each backend with a probe. The options are

**url** What URL should varnish request.

**interval** How often should we poll

**timeout** What is the timeout of the probe

**window** Varnish will maintain a *sliding window* of the results. Here the window has five checks.

**threshold** How many of the .window last polls must be good for the backend to be declared healthy.

**initial** How many of the of the probes a good when Varnish starts - defaults to the same amount as the threshold.

Now we define the director.:

```
director example_director round-robin {
      {
              .backend = server1;
      }
      # server2
      {
              .backend = server2;
      }

      }
```

You use this director just as you would use any other director or backend. Varnish will not send traffic to hosts that are marked as unhealthy. Varnish can also serve stale content if all the backends are down. See *Misbehaving servers* for more information on how to enable this.

Please note that Varnish will keep probes active for all loaded VCLs. Varnish will coalesce probes that seem identical - so be careful not to change the probe config if you do a lot of VCL loading. Unloading the VCL will discard the probes.

## 2.18 Misbehaving servers

A key feature of Varnish is its ability to shield you from misbehaving web- and application servers.

### 2.18.1 Grace mode

When several clients are requesting the same page Varnish will send one request to the backend and place the others on hold while fetching one copy from the back end. In some products this is called request coalescing and Varnish does this automatically.

If you are serving thousands of hits per second the queue of waiting requests can get huge. There are two potential problems - one is a thundering herd problem - suddenly releasing a thousand threads to serve content might send the load sky high. Secondly - nobody likes to wait. To deal with this we can instruct Varnish to keep the objects in cache beyond their TTL and to serve the waiting requests somewhat stale content.

So, in order to serve stale content we must first have some content to serve. So to make Varnish keep all objects for 30 minutes beyond their TTL use the following VCL::

```
sub vcl_fetch {
  set beresp.grace = 30m;
}
```

Varnish still won't serve the stale objects. In order to enable Varnish to actually serve the stale object we must enable this on the request. Lets us say that we accept serving 15s old object.:

```
sub vcl_recv {
  set req.grace = 15s;
}
```

You might wonder why we should keep the objects in the cache for 30 minutes if we are unable to serve them? Well, if you have enabled *Health checks* you can check if the backend is sick and if it is we can serve the stale content for a bit longer.:

```
if (! req.backend.healthy) {
   set req.grace = 5m;
} else {
   set req.grace = 15s;
}
```

**So, to sum up, grace mode solves two problems:**

- it serves stale content to avoid request pile-up.
- it serves stale content if the backend is not healthy.

### 2.18.2 Saint mode

Sometimes servers get flaky. They start throwing out random errors. You can instruct Varnish to try to handle this in a more-than-graceful way - enter *Saint mode*. Saint mode enables you to discard a certain page from one backend server and either try another server or serve stale content from cache. Lets have a look at how this can be enabled in VCL::

```
sub vcl_fetch {
  if (beresp.status == 500) {
    set beresp.saintmode = 10s;
    restart;
  }
  set beresp.grace = 5m;
}
```

When we set beresp.saintmode to 10 seconds Varnish will not ask *that* server for URL for 10 seconds. A blacklist, more or less. Also a restart is performed so if you have other backends capable of serving that content Varnish will try those. When you are out of backends Varnish will serve the content from its stale cache.

This can really be a life saver.

### 2.18.3 Known limitations on grace- and saint mode

If your request fails while it is being fetched you're thrown into vcl_error. vcl_error has access to a rather limited set of data so you can't enable saint mode or grace mode here. This will be addressed in a future release but a work-around available.

- Declare a backend that is always sick.
- Set a magic marker in vcl_error
- Restart the transaction
- Note the magic marker in vcl_recv and set the backend to the one mentioned
- Varnish will now serve stale data is any is available

### 2.18.4 God mode

Not implemented yet. :-)

## 2.19 Advanced topics

This tutorial has covered the basics in Varnish. If you read through it all you should now have the skills to run Varnish.

Here is a short overview of topics that we haven't covered in the tutorial.

### 2.19.1 More VCL

VCL is a bit more complex then what we've covered so far. There are a few more subroutines available and there a few actions that we haven't discussed. For a complete(ish) guide to VCL have a look at the VCL man page - ref:*reference-vcl*.

### 2.19.2 Using In-line C to extend Varnish

You can use *in-line C* to extend Varnish. Please note that you can seriously mess up Varnish this way. The C code runs within the Varnish Cache process so if your code generates a segfault the cache will crash.

One of the first uses I saw of In-line C was logging to syslog.:

```
# The include statements must be outside the subroutines.
C{
        #include <syslog.h>
}C

sub vcl_something {
        C{
                syslog(LOG_INFO, "Something happened at VCL line XX.");
        }C
}
```

### 2.19.3 Edge Side Includes

Varnish can cache create web pages by putting different pages together. These *fragments* can have individual cache policies. If you have a web site with a list showing the 5 most popular articles on your site, this list can probably be cached as a fragment and included in all the other pages. Used properly it can dramatically increase your hit rate and reduce the load on your servers. ESI looks like this:

```
<HTML>
<BODY>
The time is: <esi:include src="/cgi-bin/date.cgi"/>
at this very moment.
</BODY>
</HTML>
```

ESI is processed in vcl_fetch by setting *do_esi* to true.:

```
sub vcl_fetch {
    if (req.url == "/test.html") {
        set beresp.do_esi = true;  /* Do ESI processing */
    }
}
```

## 2.20 Troubleshooting Varnish

Sometimes Varnish misbehaves. In order for you to understand whats going on there are a couple of places you can check. varnishlog, /var/log/syslog, /var/log/messages are all places where varnish might leave clues of whats going on.

### 2.20.1 When Varnish won't start

Sometimes Varnish wont start. There is a plethora of reasons why Varnish wont start on your machine. We've seen everything from wrong permissions on /dev/null to other processes blocking the ports.

Starting Varnish in debug mode to see what is going on.

Try to start varnish by:

```
# varnishd -f /usr/local/etc/varnish/default.vcl -s malloc,1G -T 127.0.0.1:2000  -a 0.0.0.0:8080 -d
```

Notice the -d option. It will give you some more information on what is going on. Let us see how Varnish will react to something else listening on its port.:

```
# varnishd -n foo -f /usr/local/etc/varnish/default.vcl -s malloc,1G -T 127.0.0.1:2000  -a 0.0.0.0:8
storage_malloc: max size 1024 MB.
Using old SHMFILE
Platform: Linux,2.6.32-21-generic,i686,-smalloc,-hcritbit
200 193
-----------------------------
Varnish Cache CLI.
-----------------------------
Type 'help' for command list.
Type 'quit' to close CLI session.
Type 'start' to launch worker process.
```

Now Varnish is running. Only the master process is running, in debug mode the cache does not start. Now you're on the console. You can instruct the master process to start the cache by issuing “start”.:

```
start
bind(): Address already in use
300 22
Could not open sockets
```

And here we have our problem. Something else is bound to the HTTP port of Varnish. If this doesn't help try strace or truss or come find us on IRC.

### 2.20.2 Varnish is crashing

When varnish goes bust the child processes crashes. Usually the mother process will manage this by restarting the child process again. Any errors will be logged in syslog. It might look like this::

```
Mar  8 13:23:38 smoke varnishd[15670]: Child (15671) not responding to CLI, killing it.
Mar  8 13:23:43 smoke varnishd[15670]: last message repeated 2 times
Mar  8 13:23:43 smoke varnishd[15670]: Child (15671) died signal=3
Mar  8 13:23:43 smoke varnishd[15670]: Child cleanup complete
Mar  8 13:23:43 smoke varnishd[15670]: child (15697) Started
```

Specifically if you see the "Error in munmap" error on Linux you might want to increase the amount of maps available. Linux is limited to a maximum of 64k maps. Setting vm.max_max_count i sysctl.conf will enable you to increase this limit. You can inspect the number of maps your program is consuming by counting the lines in /proc/$PID/maps

This is a rather odd thing to document here - but hopefully Google will serve you this page if you ever encounter this error.

### 2.20.3 Varnish gives me Guru meditation

First find the relevant log entries in varnishlog. That will probably give you a clue. Since varnishlog logs so much data it might be hard to track the entries down. You can set varnishlog to log all your 503 errors by issuing the following command::

```
$ varnishlog -c -m TxStatus:503
```

If the error happened just a short time ago the transaction might still be in the shared memory log segment. To get varnishlog to process the whole shared memory log just add the -d option::

```
$ varnishlog -d -c -m TxStatus:503
```

Please see the varnishlog man page for elaborations on further filtering capabilities and explanation of the various options.

### 2.20.4 Varnish doesn't cache

See *Achieving a high hitrate*.

CHAPTER

THREE

# THE VARNISH REFERENCE MANUAL

## 3.1 VCL

### 3.1.1 Varnish Configuration Language

**Author** Dag-Erling Smørgrav

**Author** Poul-Henning Kamp

**Author** Kristian Lyngstøl

**Author** Per Buer

**Date** 2010-06-02

**Version** 1.0

**Manual section** 7

#### DESCRIPTION

The VCL language is a small domain-specific language designed to be used to define request handling and document caching policies for Varnish Cache.

When a new configuration is loaded, the varnishd management process translates the VCL code to C and compiles it to a shared object which is then dynamically linked into the server process.

#### SYNTAX

The VCL syntax is very simple, and deliberately similar to 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.

In addition to the C-like assignment (=), comparison (==, !=) and boolean (!, && and ||) operators, VCL supports both regular expression and ACL matching using the ~ and the !~ operators.

Basic strings are enclosed in ” ... ”, and may not contain newlines.

Long strings are enclosed in {” ... “}. They may contain any character including ”, newline and other control characters except for the NUL (0x00) character.

Unlike C and Perl, the backslash () character has no special meaning in strings in VCL, so it can be freely used in regular expressions without doubling.

Strings are concatenated using the ‘+’ operator.

Assignments are introduced with the *set* keyword. There are no user-defined variables; values can only be assigned to variables attached to backend, request or document objects. Most of these are typed, and the values assigned to them must have a compatible unit suffix.

You can use the *set* keyword to arbitrary HTTP headers. You can remove headers with the *remove* or *unset* keywords, which are synonym.

You can use the *rollback* keyword to revert any changes to req at any time.

The *synthetic* keyword is used to produce a synthetic response body in vcl_error. It takes a single string as argument.

You can force a crash of the client process with the *panic* keyword. *panic* takes a string as argument.

The `return(action)` keyword terminates the subroutine. *action* can be, depending on context one of

- deliver
- error
- fetch
- hash
- hit_for_pass
- lookup
- ok
- pass
- pipe
- restart

Please see the list of subroutines to see what return actions are available where.

VCL has if tests, but no loops.

The contents of another VCL file may be inserted at any point in the code by using the *include* keyword followed by the name of the other file as a quoted string.

### Backend declarations

A backend declaration creates and initializes a named backend object::

```
backend www {
  .host = "www.example.com";
  .port = "http";
}
```

The backend object can later be used to select a backend at request time::

```
if (req.http.host ~ "(?i)^(www.)?example.com$") {
  set req.backend = www;
}
```

To avoid overloading backend servers, .max_connections can be set to limit the maximum number of concurrent backend connections.

The timeout parameters can be overridden in the backend declaration. The timeout parameters are .connect_timeout for the time to wait for a backend connection, .first_byte_timeout for the time to wait for the first byte from the backend and .between_bytes_timeout for time to wait between each received byte.

These can be set in the declaration like this::

```
backend www {
  .host = "www.example.com";
  .port = "http";
  .connect_timeout = 1s;
  .first_byte_timeout = 5s;
  .between_bytes_timeout = 2s;
}
```

To mark a backend as unhealthy after number of items have been added to its saintmode list `.saintmode_threshold` can be set to the maximum list size. Setting a value of 0 disables saint mode checking entirely for that backend. The value in the backend declaration overrides the parameter.

### Directors

A director is a logical group of backend servers clustered together for redundancy. The basic role of the director is to let Varnish choose a backend server amongst several so if one is down another can be used.

There are several types of directors. The different director types use different algorithms to choose which backend to use.

Configuring a director may look like this::

```
director b2 random {
  .retries = 5;
  {
    // We can refer to named backends
    .backend = b1;
    .weight  = 7;
  }
  {
    // Or define them inline
    .backend  = {
      .host = "fs2";
    }
  .weight         = 3;
  }
}
```

**The family of random directors** There are three directors that share the same logic, called the random director, client director and hash director. They each distribute traffic among the backends assigned to it using a random distribution seeded with either the client identity, a random number or the cache hash (typically url). Beyond the initial seed, they act the same.

Each backend requires a .weight option which sets the amount of traffic each backend will get compared to the others. Equal weight means equal traffic. A backend with lower weight than an other will get proportionally less traffic.

The director has an optional .retries option which defaults to the number of backends the director has. The director will attempt .retries times to find a healthy backend if the first attempt fails. Each attempt re-uses the previous seed in an iterative manner. For the random director this detail is of no importance as it will give different results each time. For the hash and client director, this means the same URL or the same client will fail to the same server consistently.

**The random director** This uses a random number to seed the backend selection.

**The client director** The client director picks a backend based on the clients *identity*. You can set the VCL variable *client.identity* to identify the client by picking up the value of a session cookie or similar.

**The hash director** The hash director will pick a backend based on the URL hash value.

This is useful is you are using Varnish to load balance in front of other Varnish caches or other web accelerators as objects won't be duplicated across caches.

It will use the value of req.hash, just as the normal cache-lookup methods.

**The round-robin director** The round-robin director does not take any options.

It will use the first backend for the first request, the second backend for the second request and so on, and start from the top again when it gets to the end.

If a backend is unhealthy or Varnish fails to connect, it will be skipped. The round-robin director will try all the backends once before giving up.

**The DNS director** The DNS director can use backends in two different ways. Either like the random or round-robin director or using .list:

```
director directorname dns {
        .list = {
                .host_header = "www.example.com";
                .port = "80";
                .connect_timeout = 0.4s;
                "192.168.15.0"/24;
                "192.168.16.128"/25;
        }
        .ttl = 5m;
        .suffix = "internal.example.net";
}
```

This will specify 384 backends, all using port 80 and a connection timeout of 0.4s. Options must come before the list of IPs in the .list statement. The .list-method does not support IPv6. It is not a white-list, it is an actual list of backends that will be created internally in Varnish - the larger subnet the more overhead.

The .ttl defines the cache duration of the DNS lookups.

The above example will append "internal.example.net" to the incoming Host header supplied by the client, before looking it up. All settings are optional.

Health checks are not thoroughly supported.

DNS round robin balancing is supported. If a hostname resolves to multiple backends, the director will divide the traffic between all of them in a round-robin manner.

**The fallback director** The fallback director will pick the first backend that is healthy. It considers them in the order in which they are listed in its definition.

The fallback director does not take any options.

An example of a fallback director:

```
director b3 fallback {
  { .backend = www1; }
  { .backend = www2; } // will only be used if www1 is unhealthy.
  { .backend = www3; } // will only be used if both www1 and www2
                       // are unhealthy.
}
```

### Backend probes

Backends can be probed to see whether they should be considered healthy or not. The return status can also be checked by using req.backend.healthy.

Probes take the following parameters:

**.url** Specify a URL to request from the backend. Defaults to "/".

**.request** Specify a full HTTP request using multiple strings. .request will have \r\n automatically inserted after every string. If specified, .request will take precedence over .url.

**.window** How many of the latest polls we examine to determine backend health. Defaults to 8.

**.threshold** How many of the polls in .window must have succeeded for us to consider the backend healthy. Defaults to 3.

**.initial** How many of the probes are considered good when Varnish starts. Defaults to the same amount as the threshold.

**.expected_response** The expected backend HTTP response code. Defaults to 200.

**.interval** Defines how often the probe should check the backend. Default is every 5 seconds.

**.timeout** How fast each probe times out. Default is 2 seconds.

A backend with a probe can be defined like this, together with the backend or director::

```
backend www {
  .host = "www.example.com";
  .port = "http";
  .probe = {
    .url = "/test.jpg";
    .timeout = 0.3 s;
    .window = 8;
    .threshold = 3;
    .initial = 3;
  }
}
```

Or it can be defined separately and then referenced::

```
probe healthcheck {
   .url = "/status.cgi";
   .interval = 60s;
   .timeout = 0.3 s;
   .window = 8;
   .threshold = 3;
   .initial = 3;
   .expected_response = 200;
}

backend www {
  .host = "www.example.com";
  .port = "http";
  .probe = healthcheck;
}
```

If you have many backends this can simplify the config a lot.

It is also possible to specify the raw HTTP request:

```
probe rawprobe {
    # NB: \r\n automatically inserted after each string!
    .request =
      "GET / HTTP/1.1"
      "Host: www.foo.bar"
      "Connection: close";
}
```

### ACLs

An ACL declaration creates and initializes a named access control list which can later be used to match client addresses::

```
acl local {
  "localhost";         // myself
  "192.0.2.0"/24;      // and everyone on the local network
  ! "192.0.2.23";      // except for the dialin router
}
```

If an ACL entry specifies a host name which Varnish is unable to resolve, it will match any address it is compared to. Consequently, if it is preceded by a negation mark, it will reject any address it is compared to, which may not be what you intended. If the entry is enclosed in parentheses, however, it will simply be ignored.

To match an IP address against an ACL, simply use the match operator::

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

### Regular Expressions

In Varnish 2.1.0 Varnish switched to using PCRE - Perl-compatible regular expressions. For a complete description of PCRE please see the PCRE(3) man page.

To send flags to the PCRE engine, such as to turn on *case insensitivity* add the flag within parens following a question mark, like this::

```
if (req.http.host ~ "(?i)example.com$") {
        ...
}
```

### Functions

The following built-in functions are available:

**hash_data(str)** Adds a string to the hash input. In default.vcl hash_data() is called on the host and URL of the *request*.

**regsub(str, regex, sub)** Returns a copy of str with the first occurrence of the regular expression regex replaced with sub. Within sub, 0 (which can also be spelled &) is replaced with the entire matched string, and n is replaced with the contents of subgroup n in the matched string.

**regsuball(str, regex, sub)** As regsuball() but this replaces all occurrences.

ban(ban expression)

**ban_url(regex)** Bans all objects in cache whose URLs match regex.

**Subroutines** A subroutine is used to group code for legibility or reusability::

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

Subroutines in VCL do not take arguments, nor do they return values.

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

call pipe_if_local;

There are a number of special subroutines which hook into the Varnish workflow. These subroutines may inspect and manipulate HTTP headers and various other aspects of each request, and to a certain extent decide how the request should be handled. Each subroutine terminates by calling one of a small number of keywords which indicates the desired outcome.

**vcl_init** Called when VCL is loaded, before any requests pass through it. Typically used to initialize VMODs.

return() values:

**ok** Normal return, VCL continues loading.

**vcl_recv** Called at the beginning of a request, after the complete request has been received and parsed. Its purpose is to decide whether or not to serve the request, how to do it, and, if applicable, which backend to use.

The vcl_recv subroutine may terminate with calling return() on one of the following keywords:

**error code [reason]** Return the specified error code to the client and abandon the request.

**pass** Switch to pass mode. Control will eventually pass to vcl_pass.

**pipe** Switch to pipe mode. Control will eventually pass to vcl_pipe.

**lookup** Look up the requested object in the cache. Control will eventually pass to vcl_hit or vcl_miss, depending on whether the object is in the cache.

**vcl_pipe** Called upon entering pipe mode. In this mode, the request is passed on to the backend, and any further data from either client or backend is passed on unaltered until either end closes the connection.

The vcl_pipe subroutine may terminate with calling return() with one of the following keywords:

**error code [reason]** Return the specified error code to the client and abandon the request.

**pipe** Proceed with pipe mode.

**vcl_pass** Called upon entering pass mode. In this mode, the request is passed on to the backend, and the backend's response is passed on to the client, but is not entered into the cache. Subsequent requests submitted over the same client connection are handled normally.

The vcl_recv subroutine may terminate with calling return() with one of the following keywords:

**error code [reason]** Return the specified error code to the client and abandon the request.

**pass** Proceed with pass mode.

**restart** Restart the transaction. Increases the restart counter. If the number of restarts is higher than *max_restarts* varnish emits a guru meditation error.

**vcl_hash** You may call hash_data() on the data you would like to add to the hash.

The vcl_hash subroutine may terminate with calling return() with one of the following keywords:

**hash** Proceed.

**vcl_hit** Called after a cache lookup if the requested document was found in the cache.

The vcl_hit subroutine may terminate with calling return() with one of the following keywords:

**deliver** Deliver the cached object to the client. Control will eventually pass to vcl_deliver.

**error code [reason]** Return the specified error code to the client and abandon the request.

**pass** Switch to pass mode. Control will eventually pass to vcl_pass.

**restart** Restart the transaction. Increases the restart counter. If the number of restarts is higher than *max_restarts* varnish emits a guru meditation error.

**vcl_miss** Called after a cache lookup if the requested document was not found in the cache. Its purpose is to decide whether or not to attempt to retrieve the document from the backend, and which backend to use.

The vcl_miss subroutine may terminate with calling return() with one of the following keywords:

**error code [reason]** Return the specified error code to the client and abandon the request.

**pass** Switch to pass mode. Control will eventually pass to vcl_pass.

**fetch** Retrieve the requested object from the backend. Control will eventually pass to vcl_fetch.

**vcl_fetch** Called after a document has been successfully retrieved from the backend.

The vcl_fetch subroutine may terminate with calling return() with one of the following keywords:

**deliver** Possibly insert the object into the cache, then deliver it to the client. Control will eventually pass to vcl_deliver.

**error code [reason]** Return the specified error code to the client and abandon the request.

**hit_for_pass** Pass in fetch. This will create a hit_for_pass object. Note that the TTL for the hit_for_pass object will be set to what the current value of beresp.ttl. Control will be handled to vcl_deliver on the current request, but subsequent requests will go directly to vcl_pass based on the hit_for_pass object.

**restart** Restart the transaction. Increases the restart counter. If the number of restarts is higher than *max_restarts* varnish emits a guru meditation error.

**vcl_deliver** Called before a cached object is delivered to the client.

The vcl_deliver subroutine may terminate with one of the following keywords:

**deliver** Deliver the object to the client.

**error code [reason]** Return the specified error code to the client and abandon the request.

**restart** Restart the transaction. Increases the restart counter. If the number of restarts is higher than *max_restarts* varnish emits a guru meditation error.

**vcl_error** Called when we hit an error, either explicitly or implicitly due to backend or internal errors.

The vcl_error subroutine may terminate by calling return with one of the following keywords:

**deliver** Deliver the error object to the client.

**restart** Restart the transaction. Increases the restart counter. If the number of restarts is higher than *max_restarts* varnish emits a guru meditation error.

**vcl_fini** Called when VCL is discarded only after all requests have exited the VCL. Typically used to clean up VMODs.

return() values:

**ok** Normal return, VCL will be discarded.

If one of these subroutines is left undefined or terminates without reaching a handling decision, control will be handed over to the builtin default. See the EXAMPLES section for a listing of the default code.

**Multiple subroutines** If multiple subroutines with the same name are defined, they are concatenated in the order in which the appear in the source.

Example::

```
# in file "main.vcl"
include "backends.vcl";
include "ban.vcl";

# in file "backends.vcl"
sub vcl_recv {
  if (req.http.host ~ "(?i)example.com") {
    set req.backend = foo;
  } elsif (req.http.host ~ "(?i)example.org") {
    set req.backend = bar;
  }
}

# in file "ban.vcl"
sub vcl_recv {
  if (client.ip ~ admin_network) {
    if (req.http.Cache-Control ~ "no-cache") {
      ban_url(req.url);
    }
  }
}
```

The builtin default subroutines are implicitly appended in this way.

**Variables** Although subroutines take no arguments, the necessary information is made available to the handler subroutines through global variables.

The following variables are always available:

**now** The current time, in seconds since the epoch.

The following variables are available in backend declarations:

**.host** Host name or IP address of a backend.

**.port** Service name or port number of a backend.

The following variables are available while processing a request:

**client.ip** The client's IP address.

**client.identity** Identification of the client, used to load balance in the client director.

**server.hostname** The host name of the server.

**server.identity** The identity of the server, as set by the -i parameter. If the -i parameter is not passed to varnishd, server.identity will be set to the name of the instance, as specified by the -n parameter.

**server.ip** The IP address of the socket on which the client connection was received.

**server.port** The port number of the socket on which the client connection was received.

**req.request** The request type (e.g. "GET", "HEAD").

**req.url** The requested URL.

**req.proto** The HTTP protocol version used by the client.

**req.backend** The backend to use to service the request.

**req.backend.healthy** Whether the backend is healthy or not. Requires an active probe to be set on the backend.

**req.http.header** The corresponding HTTP header.

**req.hash_always_miss** Force a cache miss for this request. If set to true Varnish will disregard any existing objects and always (re)fetch from the backend.

**req.hash_ignore_busy** Ignore any busy object during cache lookup. You would want to do this if you have two server looking up content from each other to avoid potential deadlocks.

**req.can_gzip** Does the client accept the gzip transfer encoding.

**req.restarts** A count of how many times this request has been restarted.

**req.esi** Boolean. Set to false to disable ESI processing regardless of any value in beresp.do_esi. Defaults to true. This variable is subject to change in future versions, you should avoid using it.

**req.esi_level** A count of how many levels of ESI requests we're currently at.

**req.grace** Set to a period to enable grace.

**req.xid** Unique ID of this request.

The following variables are available while preparing a backend request (either for a cache miss or for pass or pipe mode):

**bereq.request** The request type (e.g. "GET", "HEAD").

**bereq.url** The requested URL.

**bereq.proto** The HTTP protocol version used to talk to the server.

**bereq.http.header** The corresponding HTTP header.

**bereq.connect_timeout** The time in seconds to wait for a backend connection.

**bereq.first_byte_timeout** The time in seconds to wait for the first byte from the backend. Not available in pipe mode.

**bereq.between_bytes_timeout** The time in seconds to wait between each received byte from the backend. Not available in pipe mode.

The following variables are available after the requested object has been retrieved from the backend, before it is entered into the cache. In other words, they are available in vcl_fetch:

**beresp.do_stream** Deliver the object to the client directly without fetching the whole object into varnish. If this request is pass'ed it will not be stored in memory. As of Varnish Cache 3.0 the object will marked as busy as it is delivered so only client can access the object.

**beresp.do_esi** Boolean. ESI-process the object after fetching it. Defaults to false. Set it to true to parse the object for ESI directives. Will only be honored if req.esi is true.

**beresp.do_gzip** Boolean. Gzip the object before storing it. Defaults to false.

**beresp.do_gunzip** Boolean. Unzip the object before storing it in the cache. Defaults to false.

**beresp.proto** The HTTP protocol version used the backend replied with.

**beresp.status** The HTTP status code returned by the server.

**beresp.response** The HTTP status message returned by the server.

**beresp.ttl** The object's remaining time to live, in seconds. beresp.ttl is writable.

**beresp.grace** Set to a period to enable grace.

**beresp.saintmode** Set to a period to enable saint mode.

**beresp.backend.name** Name of the backend this response was fetched from.

**beresp.backend.ip** IP of the backend this response was fetched from.

**beresp.backend.port** Port of the backend this response was fetched from.

**beresp.storage** Set to force Varnish to save this object to a particular storage backend.

After the object is entered into the cache, the following (mostly read-only) variables are available when the object has been located in cache, typically in vcl_hit and vcl_deliver.

**obj.proto** The HTTP protocol version used when the object was retrieved.

**obj.status** The HTTP status code returned by the server.

**obj.response** The HTTP status message returned by the server.

**obj.ttl** The object's remaining time to live, in seconds. obj.ttl is writable.

**obj.lastuse** The approximate time elapsed since the object was last requests, in seconds.

**obj.hits** The approximate number of times the object has been delivered. A value of 0 indicates a cache miss.

**obj.grace** The object's grace period in seconds. obj.grace is writable.

**obj.http.header** The corresponding HTTP header.

The following variables are available while determining the hash key of an object:

**req.hash** The hash key used to refer to an object in the cache. Used when both reading from and writing to the cache.

The following variables are available while preparing a response to the client:

**resp.proto** The HTTP protocol version to use for the response.

**resp.status** The HTTP status code that will be returned.

**resp.response** The HTTP status message that will be returned.

**resp.http.header** The corresponding HTTP header.

Values may be assigned to variables using the set keyword::

```
sub vcl_recv {
  # Normalize the Host: header
  if (req.http.host ~ "(?i)^(www.)?example.com$") {
    set req.http.host = "www.example.com";
  }
}
```

HTTP headers can be removed entirely using the remove keyword::

```
sub vcl_fetch {
  # Don't cache cookies
  remove beresp.http.Set-Cookie;
}
```

### Grace and saint mode

If the backend takes a long time to generate an object there is a risk of a thread pile up. In order to prevent this you can enable *grace*. This allows varnish to serve an expired version of the object while a fresh object is being generated by the backend.

The following vcl code will make Varnish serve expired objects. All object will be kept up to two minutes past their expiration time or a fresh object is generated.:

```
sub vcl_recv {
  set req.grace = 2m;
}
sub vcl_fetch {
  set beresp.grace = 2m;
}
```

Saint mode is similar to grace mode and relies on the same infrastructure but functions differently. You can add VCL code to vcl_fetch to see whether or not you *like* the response coming from the backend. If you find that the response is not appropriate you can set beresp.saintmode to a time limit and call *restart*. Varnish will then retry other backends to try to fetch the object again.

If there are no more backends or if you hit *max_restarts* and we have an object that is younger than what you set beresp.saintmode to be Varnish will serve the object, even if it is stale.

## EXAMPLES

The following code is the equivalent of the default configuration with the backend address set to "backend.example.com" and no backend port specified::

```
backend default {
 .host = "backend.example.com";
 .port = "http";
}

/*-
 * Copyright (c) 2006 Verdens Gang AS
 * Copyright (c) 2006-2011 Varnish Software AS
 * All rights reserved.
 *
 * Author: Poul-Henning Kamp <phk@phk.freebsd.dk>
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL AUTHOR OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * The default VCL code.
 *
 * NB! You do NOT need to copy & paste all of these functions into your
 * own vcl code, if you do not provide a definition of one of these
 * functions, the compiler will automatically fall back to the default
 * code from this file.
 *
 * This code will be prefixed with a backend declaration built from the
 * -b argument.
 */

sub vcl_recv {
    if (req.restarts == 0) {
        if (req.http.x-forwarded-for) {
            set req.http.X-Forwarded-For =
                req.http.X-Forwarded-For + ", " + client.ip;
        } else {
            set req.http.X-Forwarded-For = client.ip;
        }
    }
    if (req.request != "GET" &&
      req.request != "HEAD" &&
      req.request != "PUT" &&
      req.request != "POST" &&
      req.request != "TRACE" &&
      req.request != "OPTIONS" &&
      req.request != "DELETE") {
        /* Non-RFC2616 or CONNECT which is weird. */
        return (pipe);
    }
    if (req.request != "GET" && req.request != "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 (lookup);
}

sub vcl_pipe {
    # Note that only the first request to the backend will have
    # X-Forwarded-For set.  If you use X-Forwarded-For and want to
    # have it set for all requests, make sure to have:
    # set bereq.http.connection = "close";
    # here.  It is not set by default as it might break some broken web
    # applications, like IIS with NTLM authentication.
    return (pipe);
}

sub vcl_pass {
    return (pass);
}

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

sub vcl_hit {
    return (deliver);
}

sub vcl_miss {
    return (fetch);
}

sub vcl_fetch {
    if (beresp.ttl <= 0s ||
        beresp.http.Set-Cookie ||
        beresp.http.Vary == "*") {
                /*
                 * Mark as "Hit-For-Pass" for the next 2 minutes
                 */
                set beresp.ttl = 120 s;
                return (hit_for_pass);
    }
    return (deliver);
}

sub vcl_deliver {
    return (deliver);
}

sub vcl_error {
    set obj.http.Content-Type = "text/html; charset=utf-8";
    set obj.http.Retry-After = "5";
    synthetic {"
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <title>"} + obj.status + " " + obj.response + {"</title>
  </head>
  <body>
    <h1>Error "} + obj.status + " " + obj.response + {"</h1>
    <p>"} + obj.response + {"</p>
    <h3>Guru Meditation:</h3>
    <p>XID: "} + req.xid + {"</p>
    <hr>
    <p>Varnish cache server</p>
  </body>
</html>
"};
    return (deliver);
}

sub vcl_init {
        return (ok);
}
```

```
sub vcl_fini {
        return (ok);
}
```

The following example shows how to support multiple sites running on separate backends in the same Varnish instance, by selecting backends based on the request URL::

```
backend www {
  .host = "www.example.com";
  .port = "80";
}

backend images {
  .host = "images.example.com";
  .port = "80";
}

sub vcl_recv {
  if (req.http.host ~ "(?i)^(www.)?example.com$") {
    set req.http.host = "www.example.com";
    set req.backend = www;
  } elsif (req.http.host ~ "(?i)^images.example.com$") {
    set req.backend = images;
  } else {
    error 404 "Unknown virtual host";
  }
}

The following snippet demonstrates how to force a minimum TTL for
all documents.  Note that this is not the same as setting the
default_ttl run-time parameter, as that only affects document for
which the backend did not specify a TTL:::

import std; # needed for std.log

sub vcl_fetch {
  if (beresp.ttl < 120s) {
    std.log("Adjusting TTL");
    set beresp.ttl = 120s;
  }
}
```

The following snippet demonstrates how to force Varnish to cache documents even when cookies are present::

```
sub vcl_recv {
  if (req.request == "GET" && req.http.cookie) {
     return(lookup);
  }
}

sub vcl_fetch {
  if (beresp.http.Set-Cookie) {
     return(deliver);
 }
}
```

The following code implements the HTTP PURGE method as used by Squid for object invalidation::

```
acl purge {
  "localhost";
  "192.0.2.1"/24;
}

sub vcl_recv {
  if (req.request == "PURGE") {
    if (!client.ip ~ purge) {
      error 405 "Not allowed.";
    }
    return(lookup);
  }
}

sub vcl_hit {
  if (req.request == "PURGE") {
    purge;
    error 200 "Purged.";
  }
}

sub vcl_miss {
  if (req.request == "PURGE") {
    purge;
    error 200 "Purged.";
  }
}
```

#### SEE ALSO

- varnishd(1)
- vmod_std(7)

#### HISTORY

VCL was developed by Poul-Henning Kamp in cooperation with Verdens Gang AS, Redpill Linpro and Varnish Software. This manual page was written by Dag-Erling Smørgrav and later edited by Poul-Henning Kamp and Per Buer.

#### COPYRIGHT

This document is licensed under the same license as Varnish itself. See LICENSE for details.

- Copyright (c) 2006 Verdens Gang AS
- Copyright (c) 2006-2011 Varnish Software AS

## 3.2 varnish

### 3.2.1 Varnish Command Line Interface

**Author** Per Buer

**Date** 2011-03-23

**Version** 0.1

**Manual section** 7

## DESCRIPTION

Varnish as a command line interface (CLI) which can control and change most of the operational parameters and the configuration of Varnish, without interrupting the running service.

The CLI can be used for the following tasks:

**configuration** You can upload, change and delete VCL files from the CLI.

**parameters** You can inspect and change the various parameters Varnish has available through the CLI. The individual parameters are documented in the varnishd(1) man page.

**statistics** Statistic counters are available from the CLI.

**bans** Bans are filters that are applied to keep Varnish from serving stale content. When you issue a ban Varnish will not serve any *banned* object from cache, but rather re-fetch it from its backend servers.

**process management** You can stop and start the cache (child) process though the CLI. You can also retrieve the lastst stack trace if the child process has crashed.

If you invoke varnishd(1) with -T, -M or -d the CLI will be available. In debug mode (-d) the CLI will be in the foreground, with -T you can connect to it with varnishadm or telnet and with -M varnishd will connect back to a listening service *pushing* the CLI to that service. Please see varnishd(1) for details.

### Syntax

Commands are usually terminated with a newline. Long command can be entered using sh style *here documents*. The format of here-documents is::

```
<< word
     here document
word
```

*word* can be any continuous string choosen to make sure it doesn't appear naturally in the following *here document*.

When using the here document style of input there are no restrictions on lenght. When using newline-terminated commands maximum lenght is limited by the varnishd parameter *cli_buffer*.

When commands are newline terminated they get *tokenized* before parsing so if you have significant spaces enclose your strings in double quotes. Within the quotes you can escape characters with \. The n, r and t get translated to newlines, carrage returns and tabs. Double quotes themselves can be escaped with a backslash.

To enter characters in octals use the \nnn syntax. Hexadecimals can be entered with the \xnn syntax.

### Commands

**help [command]** Display a list of available commands.

If the command is specified, display help for this command.

**param.set param value** Set the parameter specified by param to the specified value. See Run-Time Parameters for a list of parameters.

**param.show [-l] [param]** Display a list if run-time parameters and their values.

If the -l option is specified, the list includes a brief explanation of each parameter.

If a param is specified, display only the value and explanation for this parameter.

**ping [timestamp]** Ping the Varnish cache process, keeping the connection alive.

**ban** ***field operator argument*** **[&& field operator argument [...]]** Immediately invalidate all documents matching the ban expression. See *Ban Expressions* for more documentation and examples.

**ban.list** All requests for objects from the cache are matched against items on the ban list. If an object in the cache is older than a matching ban list item, it is considered "banned", and will be fetched from the backend instead.

When a ban expression is older than all the objects in the cache, it is removed from the list.

ban.list displays the ban list. The output looks something like this (broken into two lines):

0x7fea4fcb0580 1303835108.618863 131G req.http.host ~ www.myhost.com && req.url ~ /some/url

The first field is the address of the ban.

The second is the time of entry into the list, given as a high precision timestamp.

The third field describes many objects point to this ban. When an object is compared to a ban the object is marked with a reference to the newest ban it was tested against. This isn't really useful unless you're debugging.

A "G" marks that the ban is "Gone". Meaning it has been marked as a duplicate or it is no longer valid. It stays in the list for effiency reasons.

Then follows the actual ban it self.

**ban.url regexp** Immediately invalidate all documents whose URL matches the specified regular expression. Please note that the Host part of the URL is ignored, so if you have several virtual hosts all of them will be banned. Use *ban* to specify a complete ban if you need to narrow it down.

**quit** Close the connection to the varnish admin port.

**start** Start the Varnish cache process if it is not already running.

**stats** Show summary statistics.

All the numbers presented are totals since server startup; for a better idea of the current situation, use the varnishstat(1) utility.

**status** Check the status of the Varnish cache process.

**stop** Stop the Varnish cache process.

**vcl.discard configname** Discard the configuration specified by configname. This will have no effect if the specified configuration has a non-zero reference count.

**vcl.inline configname vcl** Create a new configuration named configname with the VCL code specified by vcl, which must be a quoted string.

**vcl.list** List available configurations and their respective reference counts. The active configuration is indicated with an asterisk ("*").

**vcl.load configname filename** Create a new configuration named configname with the contents of the specified file.

**vcl.show configname** Display the source code for the specified configuration.

**vcl.use configname** Start using the configuration specified by configname for all new requests. Existing requests will continue using whichever configuration was in use when they arrived.

### Ban Expressions

A ban expression consists of one or more conditions. A condition consists of a field, an operator, and an argument. Conditions can be ANDed together with "&&".

A field can be any of the variables from VCL, for instance req.url, req.http.host or obj.set-cookie.

Operators are "==" for direct comparision, "~" for a regular expression match, and ">" or "<" for size comparisons. Prepending an operator with "!" negates the expression.

The argument could be a quoted string, a regexp, or an integer. Integers can have "KB", "MB", "GB" or "TB" appended for size related fields.

### Scripting

If you are going to write a script that talks CLI to varnishd, the include/cli.h contains the relevant magic numbers.

One particular magic number to know, is that the line with the status code and length field always is exactly 13 characters long, including the NL character.

For your reference the sourcefile lib/libvarnish/cli_common.h contains the functions varnish code uses to read and write CLI response.

### Details on authentication

If the -S secret-file is given as argument to varnishd, all network CLI connections must authenticate, by proving they know the contents of that file.

The file is read at the time the auth command is issued and the contents is not cached in varnishd, so it is possible to update the file on the fly.

Use the unix file permissions to control access to the file.

An authenticated session looks like this::

```
critter phk> telnet localhost 1234
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
107 59
ixslvvxrgkjptxmcgnnsdxsvdmvfympg

Authentication required.

auth 455ce847f0073c7ab3b1465f74507b75d3dc064c1e7de3b71e00de9092fdc89a
200 193
-----------------------------
Varnish HTTP accelerator CLI.
-----------------------------
Type 'help' for command list.
Type 'quit' to close CLI session.
Type 'start' to launch worker process.
```

The CLI status of 107 indicates that authentication is necessary. The first 32 characters of the reponse text is the challenge "ixsl...mpg". The challenge is randomly generated for each CLI connection, and changes each time a 107 is emitted.

The most recently emitted challenge must be used for calculating the authenticator “455c...c89a”.

The authenticator is calculated by applying the SHA256 function to the following byte sequence:

- Challenge string
- Newline (0x0a) character.
- Contents of the secret file
- Challenge string
- Newline (0x0a) character.

and dumping the resulting digest in lower-case hex.

In the above example, the secret file contained foon and thus::

```
critter phk> cat > _
ixslvvxrgkjptxmcgnnsdxsvdmvfympg
foo
ixslvvxrgkjptxmcgnnsdxsvdmvfympg
^D
critter phk> hexdump -C _
00000000  69 78 73 6c 76 76 78 72  67 6b 6a 70 74 78 6d 63  |ixslvvxrgkjptxmc|
00000010  67 6e 6e 73 64 78 73 76  64 6d 76 66 79 6d 70 67  |gnnsdxsvdmvfympg|
00000020  0a 66 6f 6f 0a 69 78 73  6c 76 76 78 72 67 6b 6a  |.foo.ixslvvxrgkj|
00000030  70 74 78 6d 63 67 6e 6e  73 64 78 73 76 64 6d 76  |ptxmcgnnsdxsvdmv|
00000040  66 79 6d 70 67 0a                                 |fympg.|
00000046
critter phk> sha256 _
SHA256 (_) = 455ce847f0073c7ab3b1465f74507b75d3dc064c1e7de3b71e00de9092fdc89a
critter phk> openssl dgst -sha256 < _
455ce847f0073c7ab3b1465f74507b75d3dc064c1e7de3b71e00de9092fdc89a
```

The sourcefile lib/libvarnish/cli_auth.c contains a useful function which calculates the response, given an open filedescriptor to the secret file, and the challenge string.

## EXAMPLES

Simple example: All requests where req.url exactly matches the string /news are banned from the cache::

```
req.url == "/news"
```

Example: Ban all documents where the name does not end with ”.ogg”, and where the size of the object is greater than 10 megabytes::

```
req.url !~ "\.ogg$" && obj.size > 10MB
```

Example: Ban all documents where the serving host is “example.com” or “www.example.com”, and where the Set-Cookie header received from the backend contains “USERID=1663”::

```
req.http.host ~ "^(?i)(www\.)example.com$" && obj.set-cookie ~ "USERID=1663"
```

## SEE ALSO

- varnishd(1)
- vanrishadm(1)
- vcl(7)

### HISTORY

The varnish manual page was written by Per Buer in 2011. Some of the text was taken from the Varnish Cache wiki, the varnishd(7) man page or the varnish source code.

### COPYRIGHT

## 3.3 varnishadm

### 3.3.1 Control a running varnish instance

**Author** Cecilie Fritzvold

**Author** Per Buer

**Date** 2010-05-31

**Version** 0.3

**Manual section** 1

#### SYNOPSIS

varnishadm [-t timeout] [-S secret_file] [-T address:port] [-n name] [command [...]]

#### DESCRIPTION

The varnishadm utility establishes a CLI connection to varnishd either using -n *name* or using the -T and -S arguments. If -n *name* is given the location of the secret file and the address:port is looked up in shared memory. If neither is given varnishadm will look for an instance without a given name.

If a command is given, the command and arguments are sent over the CLI connection and the result returned on stdout.

If no command argument is given varnishadm will pass commands and replies between the CLI socket and stdin/stdout.

#### OPTIONS

| | |
|---|---|
| **-t timeout** | Wait no longer than this many seconds for an operation to finish. |
| **-S secret_file** | Specify the authentication secret file. This should be the same -S argument as was given to varnishd. Only processes which can read the contents of this file, will be able to authenticate the CLI connection. |
| **-T address:port** | Connect to the management interface at the specified address and port. |
| **-n name** | Connect to the instance of varnishd with this name. |

The syntax and operation of the actual CLI interface is described in the varnish-cli(7) manual page. Parameteres are described in varnishd(1) manual page.

Additionally, a summary of commands can be obtained by issuing the *help* command, and a summary of parameters can be obtained by issuing the *param.show* command.

### EXIT STATUS

If a command is given, the exit status of the varnishadm utility is zero if the command succeeded, and non-zero otherwise.

### EXAMPLES

Some ways you can use varnishadm::

```
varnishadm -T localhost:999 -S /var/db/secret vcl.use foo
echo vcl.use foo | varnishadm -T localhost:999 -S /var/db/secret
echo vcl.use foo | ssh vhost varnishadm -T localhost:999 -S /var/db/secret
```

### SEE ALSO

- varnishd(1)

### HISTORY

The varnishadm utility and this manual page were written by Cecilie Fritzvold. Converted to reStructured and updated in 2010 by Per Buer.

### COPYRIGHT

This document is licensed under the same licence as Varnish itself. See LICENCE for details.

- Copyright (c) 2007-2011 Varnish Software AS

## 3.4 varnishd

### 3.4.1 HTTP accelerator daemon

**Author** Dag-Erling Smørgrav

**Author** Stig Sandbeck Mathisen

**Author** Per Buer

**Date** 2010-05-31

**Version** 1.0

**Manual section** 1

## SYNOPSIS

**varnishd [-a address[:port]] [-b host[:port]] [-d] [-F] [-f config]** [-g group] [-h type[,options]] [-i identity] [-l shmlogsize] [-n name] [-P file] [-p param=value] [-s type[,options]] [-T address[:port]] [-t ttl] [-u user] [-V] [-w min[,max[,timeout]]]

## DESCRIPTION

The varnishd daemon accepts HTTP requests from clients, passes them on to a backend server and caches the returned documents to better satisfy future requests for the same document.

## OPTIONS

**-a address[:port][,address[:port][...]** Listen for client requests on the specified address and port. The address can be a host name (“localhost”), an IPv4 dotted-quad (“127.0.0.1”), or an IPv6 address enclosed in square brackets (“[::1]”). If address is not specified, varnishd will listen on all available IPv4 and IPv6 interfaces. If port is not specified, the default HTTP port as listed in /etc/services is used. Multiple listening addresses and ports can be specified as a whitespace- or comma-separated list.

**-b host[:port]** Use the specified host as backend server. If port is not specified, the default is 8080.

| | |
|---|---|
| **-C** | Print VCL code compiled to C language and exit. Specify the VCL file to compile with the -f option. |
| **-d** | Enables debugging mode: The parent process runs in the foreground with a CLI connection on stdin/stdout, and the child process must be started explicitly with a CLI command. Terminating the parent process will also terminate the child. |
| **-F** | Run in the foreground. |
| **-f config** | Use the specified VCL configuration file instead of the builtin default. See vcl(7) for details on VCL syntax. When no configuration is supplied varnishd will not start the cache process. |
| **-g group** | Specifies the name of an unprivileged group to which the child process should switch before it starts accepting connections. This is a shortcut for specifying the group run-time parameter. |

**-h type[,options]** Specifies the hash algorithm. See Hash Algorithms for a list of supported algorithms.

| | |
|---|---|
| **-i identity** | Specify the identity of the varnish server. This can be accessed using server.identity from VCL |
| **-l shmlogsize** | Specify size of shmlog file. Scaling suffixes like ‘k’, ‘m’ can be used up to (e)tabytes. Default is 80 Megabytes. Specifying less than 8 Megabytes is unwise. |
| **-n name** | Specify a name for this instance. Amonst other things, this name is used to construct the name of the directory in which varnishd keeps temporary files and persistent state. If the specified name begins with a forward slash, it is interpreted as the absolute path to the directory which should be used for this purpose. |
| **-P file** | Write the process’s PID to the specified file. |

**-p param=value** Set the parameter specified by param to the specified value. See Run-Time Parameters for a list of parameters. This option can be used multiple times to specify multiple parameters.

| | |
|---|---|
| **-S file** | Path to a file containing a secret used for authorizing access to the management port. |

**-s [name=]type[,options]** Use the specified storage backend. See Storage Types for a list of supported storage types. This option can be used multiple times to specify multiple storage files. You can name the different backends. Varnish will then reference that backend with the given name in logs, statistics, etc.

**-T address[:port]** Offer a management interface on the specified address and port. See Management Interface for a list of management commands.

**-M address:port** Connect to this port and offer the command line interface. Think of it as a reverse shell. When running with -M and there is no backend defined the child process (the cache) will not start initially.

| | |
|---|---|
| **-t ttl** | Specifies a hard minimum time to live for cached documents. This is a shortcut for specifying the default_ttl run-time parameter. |
| **-u user** | Specifies the name of an unprivileged user to which the child process should switch before it starts accepting connections. This is a shortcut for specifying the user run- time parameter.<br><br>If specifying both a user and a group, the user should be specified first. |
| **-V** | Display the version number and exit. |

-w min[,max[,timeout]]

Start at least min but no more than max worker threads with the specified idle timeout. This is a shortcut for specifying the thread_pool_min, thread_pool_max and thread_pool_timeout run-time parameters.

If only one number is specified, thread_pool_min and thread_pool_max are both set to this number, and thread_pool_timeout has no effect.

### Hash Algorithms

The following hash algorithms are available:

**simple_list** A simple doubly-linked list. Not recommended for production use.

**classic[,buckets]** A standard hash table. This is the default. The hash key is the CRC32 of the object's URL modulo the size of the hash table. Each table entry points to a list of elements which share the same hash key. The buckets parameter specifies the number of entries in the hash table. The default is 16383.

**critbit** A self-scaling tree structure. The default hash algorithm in 2.1. In comparison to a more traditional B tree the critbit tree is almost completely lockless.

### Storage Types

The following storage types are available:

**malloc[,size]** Storage for each object is allocated with malloc(3).

The size parameter specifies the maximum amount of memory varnishd will allocate. The size is assumed to be in bytes, unless followed by one of the following suffixes:

K, k The size is expressed in kibibytes.

M, m The size is expressed in mebibytes.

G, g The size is expressed in gibibytes.

T, t The size is expressed in tebibytes.

The default size is unlimited.

**file[,path[,size[,granularity]]]** Storage for each object is allocated from an arena backed by a file. This is the default.

The path parameter specifies either the path to the backing file or the path to a directory in which varnishd will create the backing file. The default is /tmp.

The size parameter specifies the size of the backing file. The size is assumed to be in bytes, unless followed by one of the following suffixes:

K, k The size is expressed in kibibytes.

M, m The size is expressed in mebibytes.

G, g The size is expressed in gibibytes.

T, t The size is expressed in tebibytes.

% The size is expressed as a percentage of the free space on the file system where it resides.

The default size is 50%.

If the backing file already exists, it will be truncated or expanded to the specified size.

Note that if varnishd has to create or expand the file, it will not pre-allocate the added space, leading to fragmentation, which may adversely impact performance. Pre-creating the storage file using dd(1) will reduce fragmentation to a minimum.

The granularity parameter specifies the granularity of allocation. All allocations are rounded up to this size. The size is assumed to be in bytes, unless followed by one of the suffixes described for size except for %.

The default size is the VM page size. The size should be reduced if you have many small objects.

persistent,path,size {experimental}

Persistent storage. Varnish will store objects in a file in a manner that will secure the survival of *most* of the objects in the event of a planned or unplanned shutdown of Varnish.

The path parameter specifies the path to the backing file. If the file doesn't exist Varnish will create it.

The size parameter specifies the size of the backing file. The size is assumed to be in bytes, unless followed by one of the following suffixes:

K, k The size is expressed in kibibytes.

M, m The size is expressed in mebibytes.

G, g The size is expressed in gibibytes.

T, t The size is expressed in tebibytes.

Varnish will split the file into logical *silos* and write to the silos in the manner of a circular buffer. Only one silo will be kept open at any given point in time. Full silos are *sealed*. When Varnish starts after a shutdown it will discard the content of any silo that isn't sealed.

### Transient Storage

If you name any of your storage backend "Transient" it will be used for transient (short lived) objects. By default Varnish would use an unlimited malloc backend for this.

### Management Interface

If the -T option was specified, varnishd will offer a command-line management interface on the specified address and port. The recommended way of connecting to the command-line management interface is through varnishadm(1).

The commands available are documented in varnish(7).

### Run-Time Parameters

Runtime parameters are marked with shorthand flags to avoid repeating the same text over and over in the table below. The meaning of the flags are:

**experimental** We have no solid information about good/bad/optimal values for this parameter. Feedback with experience and observations are most welcome.

**delayed** This parameter can be changed on the fly, but will not take effect immediately.

**restart** The worker process must be stopped and restarted, before this parameter takes effect.

**reload** The VCL programs must be reloaded for this parameter to take effect.

Here is a list of all parameters, current as of last time we remembered to update the manual page. This text is produced from the same text you will find in the CLI if you use the param.show command, so should there be a new parameter which is not listed here, you can find the description using the CLI commands.

Be aware that on 32 bit systems, certain default values, such as sess_workspace (=16k) and thread_pool_stack (=64k) are reduced relative to the values listed here, in order to conserve VM space.

**acceptor_sleep_decay**

- Default: 0.900
- Flags: experimental

If we run out of resources, such as file descriptors or worker threads, the acceptor will sleep between accepts. This parameter (multiplicatively) reduce the sleep duration for each succesfull accept. (ie: 0.9 = reduce by 10%)

**acceptor_sleep_incr**

- Units: s
- Default: 0.001
- Flags: experimental

If we run out of resources, such as file descriptors or worker threads, the acceptor will sleep between accepts. This parameter control how much longer we sleep, each time we fail to accept a new connection.

**acceptor_sleep_max**

- Units: s
- Default: 0.050
- Flags: experimental

If we run out of resources, such as file descriptors or worker threads, the acceptor will sleep between accepts. This parameter limits how long it can sleep between attempts to accept new connections.

**auto_restart**

- Units: bool
- Default: on

Restart child process automatically if it dies.

**ban_dups**

- Units: bool

- Default: on

Detect and eliminate duplicate bans.

**ban_lurker_sleep**

- Units: s
- Default: 0.01

How long time does the ban lurker thread sleeps between successful attempts to push the last item up the ban list. It always sleeps a second when nothing can be done. A value of zero disables the ban lurker.

**between_bytes_timeout**

- Units: s
- Default: 60

Default timeout between bytes when receiving data from backend. We only wait for this many seconds between bytes before giving up. A value of 0 means it will never time out. VCL can override this default value for each backend request and backend request. This parameter does not apply to pipe.

**cc_command**

- Default: exec gcc -std=gnu99 -pthread -fpic -shared -Wl,-x -o %o %s
- Flags: must_reload

Command used for compiling the C source code to a dlopen(3) loadable object. Any occurrence of %s in the string will be replaced with the source file name, and %o will be replaced with the output file name.

**cli_buffer**

- Units: bytes
- Default: 8192

Size of buffer for CLI input. You may need to increase this if you have big VCL files and use the vcl.inline CLI command. NB: Must be specified with -p to have effect.

**cli_timeout**

- Units: seconds
- Default: 10

Timeout for the childs replies to CLI requests from the master.

**clock_skew**

- Units: s
- Default: 10

How much clockskew we are willing to accept between the backend and our own clock.

**connect_timeout**

- Units: s
- Default: 0.7

Default connection timeout for backend connections. We only try to connect to the backend for this many seconds before giving up. VCL can override this default value for each backend and backend request.

**critbit_cooloff**

- Units: s

- Default: 180.0
- Flags:

How long time the critbit hasher keeps deleted objheads on the cooloff list.

**default_grace**

- Units: seconds
- Default: 10
- Flags: delayed

Default grace period. We will deliver an object this long after it has expired, provided another thread is attempting to get a new copy. Objects already cached will not be affected by changes made until they are fetched from the backend again.

**default_keep**

- Units: seconds
- Default: 0
- Flags: delayed

Default keep period. We will keep a useless object around this long, making it available for conditional backend fetches. That means that the object will be removed from the cache at the end of ttl+grace+keep.

**default_ttl**

- Units: seconds
- Default: 120

The TTL assigned to objects if neither the backend nor the VCL code assigns one. Objects already cached will not be affected by changes made until they are fetched from the backend again. To force an immediate effect at the expense of a total flush of the cache use "ban.url ."

**diag_bitmap**

- Units: bitmap
- Default: 0

Bitmap controlling diagnostics code:

```
0x00000001 - CNT_Session states.
0x00000002 - workspace debugging.
0x00000004 - kqueue debugging.
0x00000008 - mutex logging.
0x00000010 - mutex contests.
0x00000020 - waiting list.
0x00000040 - object workspace.
0x00001000 - do not core-dump child process.
0x00002000 - only short panic message.
0x00004000 - panic to stderr.
0x00010000 - synchronize shmlog.
0x00020000 - synchronous start of persistence.
0x00040000 - release VCL early.
0x80000000 - do edge-detection on digest.
```

Use 0x notation and do the bitor in your head :-)

**esi_syntax**

- Units: bitmap

- Default: 0

Bitmap controlling ESI parsing code:

```
0x00000001 - Don't check if it looks like XML
0x00000002 - Ignore non-esi elements
0x00000004 - Emit parsing debug records
0x00000008 - Force-split parser input (debugging)
```

Use 0x notation and do the bitor in your head :-)

**expiry_sleep**

- Units: seconds
- Default: 1

How long the expiry thread sleeps when there is nothing for it to do.

**fetch_chunksize**

- Units: kilobytes
- Default: 128
- Flags: experimental

The default chunksize used by fetcher. This should be bigger than the majority of objects with short TTLs. Internal limits in the storage_file module makes increases above 128kb a dubious idea.

**fetch_maxchunksize**

- Units: kilobytes
- Default: 262144
- Flags: experimental

The maximum chunksize we attempt to allocate from storage. Making this too large may cause delays and storage fragmentation.

**first_byte_timeout**

- Units: s
- Default: 60

Default timeout for receiving first byte from backend. We only wait for this many seconds for the first byte before giving up. A value of 0 means it will never time out. VCL can override this default value for each backend and backend request. This parameter does not apply to pipe.

**group**

- Default: magic
- Flags: must_restart

The unprivileged group to run as.

**gzip_level**

- Default: 6

Gzip compression level: 0=debug, 1=fast, 9=best

**gzip_memlevel**

- Default: 8

Gzip memory level 1=slow/least, 9=fast/most compression. Memory impact is 1=1k, 2=2k, ... 9=256k.

**gzip_stack_buffer**

- Units: Bytes
- Default: 32768
- Flags: experimental

Size of stack buffer used for gzip processing. The stack buffers are used for in-transit data, for instance gunzip'ed data being sent to a client.Making this space to small results in more overhead, writes to sockets etc, making it too big is probably just a waste of memory.

**gzip_tmp_space**

- Default: 0
- Flags: experimental

Where temporary space for gzip/gunzip is allocated:

```
0 - malloc
1 - session workspace
2 - thread workspace
```

If you have much gzip/gunzip activity, it may be an advantage to use workspace for these allocations to reduce malloc activity. Be aware that gzip needs 256+KB and gunzip needs 32+KB of workspace (64+KB if ESI processing).

**gzip_window**

- Default: 15

Gzip window size 8=least, 15=most compression. Memory impact is 8=1k, 9=2k, ... 15=128k.

**http_gzip_support**

- Units: bool
- Default: on
- Flags: experimental

Enable gzip support. When enabled Varnish will compress uncompressed objects before they are stored in the cache. If a client does not support gzip encoding Varnish will uncompress compressed objects on demand. Varnish will also rewrite the Accept-Encoding header of clients indicating support for gzip to:

Accept-Encoding: gzip

Clients that do not support gzip will have their Accept-Encoding header removed. For more information on how gzip is implemented please see the chapter on gzip in the Varnish reference.

**http_max_hdr**

- Units: header lines
- Default: 64

Maximum number of HTTP headers we will deal with in client request or backend reponses. Note that the first line occupies five header fields. This paramter does not influence storage consumption, objects allocate exact space for the headers they store.

**http_range_support**

- Units: bool
- Default: on

- Flags: experimental

Enable support for HTTP Range headers.

**http_req_hdr_len**

- Units: bytes
- Default: 8192

Maximum length of any HTTP client request header we will allow. The limit is inclusive its continuation lines.

**http_req_size**

- Units: bytes
- Default: 32768

Maximum number of bytes of HTTP client request we will deal with. This is a limit on all bytes up to the double blank line which ends the HTTP request. The memory for the request is allocated from the session workspace (param: sess_workspace) and this parameter limits how much of that the request is allowed to take up.

**http_resp_hdr_len**

- Units: bytes
- Default: 8192

Maximum length of any HTTP backend response header we will allow. The limit is inclusive its continuation lines.

**http_resp_size**

- Units: bytes
- Default: 32768

Maximum number of bytes of HTTP backend resonse we will deal with. This is a limit on all bytes up to the double blank line which ends the HTTP request. The memory for the request is allocated from the worker workspace (param: sess_workspace) and this parameter limits how much of that the request is allowed to take up.

**listen_address**

- Default: :80
- Flags: must_restart

Whitespace separated list of network endpoints where Varnish will accept requests. Possible formats: host, host:port, :port

**listen_depth**

- Units: connections
- Default: 1024
- Flags: must_restart

Listen queue depth.

**log_hashstring**

- Units: bool
- Default: on

Log the hash string components to shared memory log.

**log_local_address**

- Units: bool
- Default: off

Log the local address on the TCP connection in the SessionOpen shared memory record.

**lru_interval**

- Units: seconds
- Default: 2
- Flags: experimental

Grace period before object moves on LRU list. Objects are only moved to the front of the LRU list if they have not been moved there already inside this timeout period. This reduces the amount of lock operations necessary for LRU list access.

**max_esi_depth**

- Units: levels
- Default: 5

Maximum depth of esi:include processing.

**max_restarts**

- Units: restarts
- Default: 4

Upper limit on how many times a request can restart. Be aware that restarts are likely to cause a hit against the backend, so don't increase thoughtlessly.

**nuke_limit**

- Units: allocations
- Default: 50
- Flags: experimental

Maximum number of objects we attempt to nuke in orderto make space for a object body.

**ping_interval**

- Units: seconds
- Default: 3
- Flags: must_restart

Interval between pings from parent to child. Zero will disable pinging entirely, which makes it possible to attach a debugger to the child.

**pipe_timeout**

- Units: seconds
- Default: 60

Idle timeout for PIPE sessions. If nothing have been received in either direction for this many seconds, the session is closed.

**prefer_ipv6**

- Units: bool

- Default: off

Prefer IPv6 address when connecting to backends which have both IPv4 and IPv6 addresses.

**queue_max**

- Units: %
- Default: 100
- Flags: experimental

Percentage permitted queue length.

This sets the ratio of queued requests to worker threads, above which sessions will be dropped instead of queued.

**rush_exponent**

- Units: requests per request
- Default: 3
- Flags: experimental

How many parked request we start for each completed request on the object. NB: Even with the implict delay of delivery, this parameter controls an exponential increase in number of worker threads.

**saintmode_threshold**

- Units: objects
- Default: 10
- Flags: experimental

The maximum number of objects held off by saint mode before no further will be made to the backend until one times out. A value of 0 disables saintmode.

**send_timeout**

- Units: seconds
- Default: 60
- Flags: delayed

Send timeout for client connections. If the HTTP response hasn't been transmitted in this many seconds the session is closed. See setsockopt(2) under SO_SNDTIMEO for more information.

**sess_timeout**

- Units: seconds
- Default: 5

Idle timeout for persistent sessions. If a HTTP request has not been received in this many seconds, the session is closed.

**sess_workspace**

- Units: bytes
- Default: 65536
- Flags: delayed

Bytes of HTTP protocol workspace allocated for sessions. This space must be big enough for the entire HTTP protocol header and any edits done to it in the VCL code. Minimum is 1024 bytes.

**session_linger**

- Units: ms
- Default: 50
- Flags: experimental

How long time the workerthread lingers on the session to see if a new request appears right away. If sessions are reused, as much as half of all reuses happen within the first 100 msec of the previous request completing. Setting this too high results in worker threads not doing anything for their keep, setting it too low just means that more sessions take a detour around the waiter.

**session_max**

- Units: sessions
- Default: 100000

Maximum number of sessions we will allocate before just dropping connections. This is mostly an anti-DoS measure, and setting it plenty high should not hurt, as long as you have the memory for it.

**shm_reclen**

- Units: bytes
- Default: 255

Maximum number of bytes in SHM log record. Maximum is 65535 bytes.

**shm_workspace**

- Units: bytes
- Default: 8192
- Flags: delayed

Bytes of shmlog workspace allocated for worker threads. If too big, it wastes some ram, if too small it causes needless flushes of the SHM workspace. These flushes show up in stats as “SHM flushes due to overflow”. Minimum is 4096 bytes.

**shortlived**

- Units: s
- Default: 10.0

Objects created with TTL shorter than this are always put in transient storage.

**syslog_cli_traffic**

- Units: bool
- Default: on

Log all CLI traffic to syslog(LOG_INFO).

**thread_pool_add_delay**

- Units: milliseconds
- Default: 2

Wait at least this long between creating threads.

Setting this too long results in insuffient worker threads.

Setting this too short increases the risk of worker thread pile-up.

**thread_pool_add_threshold**

- Units: requests
- Default: 2
- Flags: experimental

Overflow threshold for worker thread creation.

Setting this too low, will result in excess worker threads, which is generally a bad idea.

Setting it too high results in insuffient worker threads.

**thread_pool_fail_delay**

- Units: milliseconds
- Default: 200
- Flags: experimental

Wait at least this long after a failed thread creation before trying to create another thread.

Failure to create a worker thread is often a sign that the end is near, because the process is running out of RAM resources for thread stacks. This delay tries to not rush it on needlessly.

If thread creation failures are a problem, check that thread_pool_max is not too high.

It may also help to increase thread_pool_timeout and thread_pool_min, to reduce the rate at which treads are destroyed and later recreated.

**thread_pool_max**

- Units: threads
- Default: 500
- Flags: delayed, experimental

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.

**thread_pool_min**

- Units: threads
- Default: 5
- Flags: delayed, experimental

The minimum number of worker threads in each pool.

Increasing this may help ramp up faster from low load situations where threads have expired.

Minimum is 2 threads.

**thread_pool_purge_delay**

- Units: milliseconds
- Default: 1000
- Flags: delayed, experimental

Wait this long between purging threads.

This controls the decay of thread pools when idle(-ish).

Minimum is 100 milliseconds.

**thread_pool_stack**

- Units: bytes
- Default: -1
- Flags: experimental

Worker thread stack size. On 32bit systems you may need to tweak this down to fit many threads into the limited address space.

**thread_pool_timeout**

- Units: seconds
- Default: 300
- Flags: delayed, experimental

Thread idle threshold.

Threads in excess of thread_pool_min, which have been idle for at least this long are candidates for purging.

Minimum is 1 second.

**thread_pool_workspace**

- Units: bytes
- Default: 65536
- Flags: delayed

Bytes of HTTP protocol workspace allocated for worker threads. This space must be big enough for the backend request and responses, and response to the client plus any other memory needs in the VCL code.Minimum is 1024 bytes.

**thread_pools**

- Units: pools
- Default: 2
- Flags: delayed, experimental

Number of worker thread pools.

Increasing number of worker pools decreases lock contention.

Too many pools waste CPU and RAM resources, and more than one pool for each CPU is probably detrimal to performance.

Can be increased on the fly, but decreases require a restart to take effect.

**thread_stats_rate**

- Units: requests
- Default: 10
- Flags: experimental

Worker threads accumulate statistics, and dump these into the global stats counters if the lock is free when they finish a request. This parameters defines the maximum number of requests a worker thread may handle, before it is forced to dump its accumulated stats into the global counters.

**user**

- Default: magic

- Flags: must_restart

The unprivileged user to run as. Setting this will also set “group” to the specified user’s primary group.

**vcc_err_unref**

- Units: bool
- Default: on

Unreferenced VCL objects result in error.

**vcl_dir**

- Default: /usr/local/etc/varnish

Directory from which relative VCL filenames (vcl.load and include) are opened.

**vcl_trace**

- Units: bool
- Default: off

Trace VCL execution in the shmlog. Enabling this will allow you to see the path each request has taken through the VCL program. This generates a lot of logrecords so it is off by default.

**vmod_dir**

- Default: /usr/local/lib/varnish/vmods

Directory where VCL modules are to be found.

**waiter**

- Default: default
- Flags: must_restart, experimental

Select the waiter kernel interface.

## SEE ALSO

- varnish-cli(7)
- varnishlog(1)
- varnishhist(1)
- varnishncsa(1)
- varnishstat(1)
- varnishtop(1)
- vcl(7)

## HISTORY

The varnishd daemon was developed by Poul-Henning Kamp in cooperation with Verdens Gang AS, Varnish Software AS and Varnish Software.

This manual page was written by Dag-Erling Smørgrav with updates by Stig Sandbeck Mathisen (ssm@debian.org)

### COPYRIGHT

## 3.5 varnishhist

### 3.5.1 Varnish request histogram

**Author** Dag-Erling Smørgrav

**Date** 2010-05-31

**Version** 1.0

**Manual section** 1

#### SYNOPSIS

varnishhist [-b] [-C] [-c] [-d] [-I regex] [-i tag] [-m tag:regex ...] [-n varnish_name] [-r file] [-V] [-w delay] [-X regex] [-x tag]

#### DESCRIPTION

The varnishhist utility reads varnishd(1) shared memory logs and presents a continuously updated histogram showing the distribution of the last N requests by their processing. The value of N and the vertical scale are displayed in the top left corner. The horizontal scale is logarithmic. Hits are marked with a pipe character ("|"), and misses are marked with a hash character ("#").

The following options are available:

| | |
|---|---|
| **-b** | Include log entries which result from communication with a backend server. If neither -b nor -c is specified, varnishhist acts as if they both were. |
| **-C** | Ignore case when matching regular expressions. |
| **-c** | Include log entries which result from communication with a client. If neither -b nor -c is specified, varnishhist acts as if they both were. |
| **-d** | Process old log entries on startup. Normally, varnishhist will only process entries which are written to the log after it starts. |
| **-I regex** | Include log entries which match the specified regular expression. If neither -I nor -i is specified, all log entries are included. |
| **-i tag** | Include log entries with the specified tag. If neither -I nor -i is specified, all log entries are included. |

**-m tag:regex only count transactions where tag matches regex. Multiple** -m options are AND-ed together.

| | |
|---|---|
| **-n** | Specifies the name of the varnishd instance to get logs from. If -n is not specified, the host name is used. |
| **-r file** | Read log entries from file instead of shared memory. |
| **-V** | Display the version number and exit. |

| | |
|---|---|
| **-w delay** | Wait at least delay seconds between each update. The default is 1. file instead of displaying them. The file will be overwritten unless the -a option was specified. |
| **-X regex** | Exclude log entries which match the specified regular expression. |
| **-x tag** | Exclude log entries with the specified tag. |

#### SEE ALSO

- varnishd(1)
- varnishlog(1)
- varnishncsa(1)
- varnishstat(1)
- varnishtop(1)

#### HISTORY

The varnishhist utility was developed by Poul-Henning Kamp in cooperation with Verdens Gang AS and Varnish Software AS. This manual page was written by Dag-Erling Smørgrav.

#### COPYRIGHT

This document is licensed under the same licence as Varnish itself. See LICENCE for details.

- Copyright (c) 2006 Verdens Gang AS
- Copyright (c) 2006-2011 Varnish Software AS

## 3.6 varnishlog

### 3.6.1 Display Varnish logs

**Author** Dag-Erling Smørgrav

**Author** Per Buer

**Date** 2010-05-31

**Version** 0.2

**Manual section** 1

#### SYNOPSIS

varnishlog [-a] [-b] [-C] [-c] [-D] [-d] [-I regex] [-i tag] [-k keep] [-n varnish_name] [-o] [-O] [-m tag:regex ...] [-P file] [-r file] [-s num] [-u] [-V] [-w file] [-X regex] [-x tag]

## DESCRIPTION

The varnishlog utility reads and presents varnishd(1) shared memory logs.

The following options are available:

| | |
|---|---|
| **-a** | When writing to a file, append to it rather than overwrite it. |
| **-b** | Include log entries which result from communication with a backend server. If neither -b nor -c is specified, varnishlog acts as if they both were. |
| **-C** | Ignore case when matching regular expressions. |
| **-c** | Include log entries which result from communication with a client. If neither -b nor -c is specified, varnishlog acts as if they both were. |
| **-D** | Daemonize. |
| **-d** | Process old log entries on startup. Normally, varnishlog will only process entries which are written to the log after it starts. |
| **-I regex** | Include log entries which match the specified regular expression. If neither -I nor -i is specified, all log entries are included. |
| **-i tag** | Include log entries with the specified tag. If neither -I nor -i is specified, all log entries are included. |
| **-k num** | Only show the first num log records. |

**-m tag:regex only list transactions where tag matches regex. Multiple** -m options are AND-ed together. Can not be combined with -O

| | |
|---|---|
| **-n** | Specifies the name of the varnishd instance to get logs from. If -n is not specified, the host name is used. |
| **-o** | Ignored for compatibility with earlier versions. |
| **-O** | Do not group log entries by request ID. Can not be combined with -m. |
| **-P file** | Write the process's PID to the specified file. |
| **-r file** | Read log entries from file instead of shared memory. |
| **-s num** | Skip the first num log records. |
| **-u** | Unbuffered output. |
| **-V** | Display the version number and exit. |
| **-w file** | Write log entries to file instead of displaying them. The file will be overwritten unless the -a option was specified. If varnishlog receives a SIGHUP while writing to a file, it will reopen the file, allowing the old one to be rotated away. |
| **-X regex** | Exclude log entries which match the specified regular expression. |
| **-x tag** | Exclude log entries with the specified tag. |

## TAGS

The following log entry tags are currently defined:

- Backend
- BackendClose
- BackendOpen

- BackendReuse
- BackendXID
- CLI
- ClientAddr
- Debug
- Error
- ExpBan
- ExpKill
- ExpPick
- Hit
- HitPass
- HttpError
- HttpGarbage
- Length
- ObjHeader
- ObjLostHeader
- ObjProtocol
- ObjRequest
- ObjResponse
- ObjStatus
- ObjURL
- ReqEnd
- ReqStart
- RxHeader
- RxLostHeader
- RxProtocol
- RxRequest
- RxResponse
- RxStatus
- RxURL
- SessionClose
- SessionOpen
- StatAddr
- StatSess
- TTL
- TxHeader

- TxLostHeader
- TxProtocol
- TxRequest
- TxResponse
- TxStatus
- TxURL
- VCL_acl
- VCL_call
- VCL_return
- VCL_trace
- WorkThread

## EXAMPLES

The following command line simply copies all log entries to a log file::

```
$ varnishlog -w /var/log/varnish.log
```

The following command line reads that same log file and displays requests for the front page::

```
$ varnishlog -r /var/log/varnish.log -c -m 'RxURL:^/$'
```

## SEE ALSO

- varnishd(1)
- varnishhist(1)
- varnishncsa(1)
- varnishstat(1)
- varnishtop(1)

## HISTORY

The varnishlog utility was developed by Poul-Henning Kamp (phk@phk.freebsd.dk) in cooperation with Verdens Gang AS, Varnish Software AS and Varnish Software. This manual page was initially written by Dag-Erling Smørgrav.

## COPYRIGHT

This document is licensed under the same licence as Varnish itself. See LICENCE for details.

- Copyright (c) 2006 Verdens Gang AS
- Copyright (c) 2006-2011 Varnish Software AS

## 3.7 varnishncsa

### 3.7.1 Display Varnish logs in Apache / NCSA combined log format

**Author** Dag-Erling Smørgrav

**Date** 2010-05-31

**Version** 1.0

**Manual section** 1

#### SYNOPSIS

varnishncsa [-a] [-b] [-C] [-c] [-D] [-d] [-f] [-F format] [-I regex] [-i tag] [-n varnish_name] [-m tag:regex ...] [-P file] [-r file] [-V] [-w file] [-X regex] [-x tag]

#### DESCRIPTION

The varnishncsa utility reads varnishd(1) shared memory logs and presents them in the Apache / NCSA "combined" log format.

The following options are available:

| | |
|---|---|
| **-a** | When writing to a file, append to it rather than overwrite it. |
| **-b** | Include log entries which result from communication with a backend server. If neither -b nor -c is specified, varnishncsa acts as if they both were. |
| **-C** | Ignore case when matching regular expressions. |
| **-c** | Include log entries which result from communication with a client. If neither -b nor -c is specified, varnishncsa acts as if they both were. |
| **-D** | Daemonize. |
| **-d** | Process old log entries on startup. Normally, varnishncsa will only process entries which are written to the log after it starts. |
| **-f** | Prefer the X-Forwarded-For HTTP header over client.ip in the log output. |
| **-F format** | Specify the log format used. If no format is specified the default log format is used. Currently it is: |

%h %l %u %t "%r" %s %b "%{Referer}i" "%{User-agent}i"

Supported formatters are:

**%b** Size of response in bytes, excluding HTTP headers. In CLF format, i.e. a '-' rather than a 0 when no bytes are sent.

**%H** The request protocol. Defaults to HTTP/1.0 if not known.

**%h** Remote host. Defaults to '-' if not known. Defaults to 127.0.0.1 for backend requests.

**%{X}i** The contents of request header line X.

**%l** Remote logname (always '-')

**%m** Request method. Defaults to '-' if not known.

**%q** The query string, if no query string exists, an empty string.

**%{X}o** The contents of response header line X.

**%r** The first line of the request. Synthesized from other fields, so it may not be the request verbatim.

**%s** Status sent to the client

**%t** Time when the request was received, in HTTP date/time format.

**%U** The request URL without any query string. Defaults to '-' if not known.

**%u** Remote user from auth

**%{X}x** Extended variables. Supported variables are:

**Varnish:time_firstbyte** Time to the first byte from the backend arrived

**Varnish:hitmiss** Whether the request was a cache hit or miss. Pipe and pass are considered misses.

**Varnish:handling** How the request was handled, whether it was a cache hit, miss, pass, pipe or error.

**-m tag:regex only list records where tag matches regex. Multiple** -m options are AND-ed together.

| | |
|---|---|
| **-n** | Specifies the name of the varnishd instance to get logs from. If -n is not specified, the host name is used. |
| **-P file** | Write the process's PID to the specified file. |
| **-r file** | Read log entries from file instead of shared memory. |
| **-V** | Display the version number and exit. |
| **-w file** | Write log entries to file instead of displaying them. The file will be overwritten unless the -a option was specified.<br>If varnishncsa receives a SIGHUP while writing to a file, it will reopen the file, allowing the old one to be rotated away. |
| **-X regex** | Exclude log entries which match the specified regular expression. |
| **-x tag** | Exclude log entries with the specified tag. |

If the -o option was specified, a tag and a regex argument must be given. varnishncsa will then only log for request groups which include that tag and the regular expression matches on that tag.

## SEE ALSO

- varnishd(1)
- varnishhist(1)
- varnishlog(1)
- varnishstat(1)
- varnishtop(1)

### HISTORY

The varnishncsa utility was developed by Poul-Henning Kamp in cooperation with Verdens Gang AS and Varnish Software AS. This manual page was written by Dag-Erling Smørgrav (des@des.no).

### COPYRIGHT

This document is licensed under the same licence as Varnish itself. See LICENCE for details.

- Copyright (c) 2006 Verdens Gang AS
- Copyright (c) 2006-2011 Varnish Software AS

## 3.8 varnishreplay

### 3.8.1 HTTP traffic replay tool

**Author** Cecilie Fritzvold

**Author** Per Buer

**Date** 2010-05-31

**Version** 1.0

**Manual section** 1

#### SYNOPSIS

varnishreplay [-D] -a address:port -r file

#### DESCRIPTION

The varnishreplay utility parses varnish logs and attempts to reproduce the traffic. It is typcally used to *warm* up caches or various forms of testing.

The following options are available:

| | |
|---|---|
| **-a backend** | Send the traffic over tcp to this server, specified by an address and a port. This option is mandatory. Only IPV4 is supported at this time. |
| **-D** | Turn on debugging mode. |
| **-r file** | Parse logs from this file. This option is mandatory. |

#### SEE ALSO

- varnishd(1)
- varnishlog(1)

#### HISTORY

The varnishreplay utility and this manual page were written by Cecilie Fritzvold and later updated by Per Buer.

### COPYRIGHT

## 3.9 varnishsizes

### 3.9.1 Varnish object size request histogram

**Author** Dag Erling Smørgrav

**Author** Kristian Lyngstøl

**Author** Per Buer

**Date** 2010-05-31

**Version** 1.0

**Manual section** 1

#### SYNOPSIS

**varnishsizes [-b] [-C] [-c] [-d] [-I regex] [-i tag] [-m tag:regex ...]** [-n varnish_name] [-r file] [-V] [-w delay] [-X regex] [-x tag]

#### DESCRIPTION

The varnishsizes utility reads varnishd(1) shared memory logs and presents a continuously updated histogram showing the distribution of the last N requests by their processing. The value of N and the vertical scale are displayed in the top left corner. The horizontal scale is a logarithmic representation of bytes. Hits are marked with a pipe character (“|”), and misses are marked with a hash character (“#”).

The following options are available:

| | |
|---|---|
| **-b** | Include log entries which result from communication with a backend server. If neither -b nor -c is specified, varnishsizes acts as if they both were. |
| **-C** | Ignore case when matching regular expressions. |
| **-c** | Include log entries which result from communication with a client. If neither -b nor -c is specified, varnishsizes acts as if they both were. |
| **-d** | Process old log entries on startup. Normally, varnishsizes will only process entries which are written to the log after it starts. |
| **-I regex** | Include log entries which match the specified regular expression. If neither -I nor -i is specified, all log entries are included. |
| **-i tag** | Include log entries with the specified tag. If neither -I nor -i is specified, all log entries are included. |

**-m tag:regex only list record where tag matches regex. Multiple -m** options are AND-ed together.

| | |
|---|---|
| **-n** | Specifies the name of the varnishd instance to get logs from. If -n is not specified, the host name is used. |
| **-r file** | Read log entries from file instead of shared memory. |

| | |
|---|---|
| **-V** | Display the version number and exit. |
| **-w delay** | Wait at least delay seconds between each update. The default is 1. file instead of displaying them. The file will be overwritten unless the -a option was specified. |
| **-X regex** | Exclude log entries which match the specified regular expression. |
| **-x tag** | Exclude log entries with the specified tag. |

#### SEE ALSO

- varnishd(1)
- varnishlog(1)
- varnishncsa(1)
- varnishstat(1)
- varnishtop(1)

#### HISTORY

The varnishsizes utility was developed by Kristian Lyngstøl based on varnishhist. This manual page was written by Kristian Lyngstøl, Dag-Erling Smørgrav and Per Buer.

#### COPYRIGHT

This document is licensed under the same licence as Varnish itself. See LICENCE for details.

- Copyright (c) 2010 Varnish Software AS

## 3.10 varnishstat

### 3.10.1 Varnish Cache statistics

**Author** Dag-Erling Smørgrav

**Author** Per Buer

**Date** 2010-06-1

**Version** 1.0

**Manual section** 1

#### SYNOPSIS

varnishstat [-1] [-x] [-f field_list] [-l] [-n varnish_name] [-V] [-w delay]

## DESCRIPTION

The varnishstat utility displays statistics from a running varnishd(1) instance.

The following options are available:

| | |
|---|---|
| **-1** | Instead of presenting of a continuously updated display, print the statistics once and exit. |
| **-f** | A comma separated list of the fields to display. If it starts with '^' it is used as an exclusion list. |
| **-l** | Lists the available fields to use with the -f option. |
| **-n** | Specifies the name of the varnishd instance to get logs from. If -n is not specified, the host name is used. |
| **-V** | Display the version number and exit. |
| **-w delay** | Wait delay seconds between updates. The default is 1. |
| **-x** | Displays the result as XML once. |

The columns in the main display are, from left to right:

1. Value
2. Per-second average in the period since last update, or a period if the value can not be averaged
3. Per-second average over process lifetime, or a period if the value can not be averaged
4. Descriptive text

When using the -1 option, the columns in the output are, from left to right:

1. Symbolic entry name
2. Value
3. Per-second average over process lifetime, or a period if the value can not be averaged
4. Descriptive text

When using the -x option, the output is:

```
<stat>
  <name>FIELD NAME</name>
  <value>FIELD VALUE</value>
  <description>FIELD DESCRIPTION</description>
</stat>
```

## SEE ALSO

- varnishd(1)
- varnishhist(1)
- varnishlog(1)
- varnishncsa(1)
- varnishtop(1)
- curses(3)

#### HISTORY

The varnishstat utility was originally developed by Poul-Henning Kamp (phk@phk.freebsd.dk) in cooperation with Verdens Gang AS, Varnish Software AS and Varnish Software. Manual page written by Dag-Erling Smørgrav, and Per Buer.

#### COPYRIGHT

## 3.11 varnishtest

### 3.11.1 Test program for Varnish

**Author** Stig Sandbeck Mathisen

**Date** 2010-05-31

**Version** 1.0

**Manual section** 1

#### SYNOPSIS

varnishtest [-n iter] [-q] [-v] file [file ...]

#### DESCRIPTION

The varnishtest program is a script driven program used to test the Varnish Cache.

The varnishtest program, when started and given one or more script files, can create a number of threads representing backends, some threads representing clients, and a varnishd process.

The following options are available:

| | |
|---|---|
| **-n iter** | Run iter number of iterations. |
| **-q** | Be quiet. |
| **-v** | Be verbose. |
| **-t** | Dunno. |

file File to use as a script

#### SCRIPTS

##### Example script

```
# Start a varnish instance called "v1"
varnish v1 -arg "-b localhost:9080" -start

# Create a server thread called "s1"
server s1 {
    # Receive a request
    rxreq
    # Send a standard response
    txresp -hdr "Connection: close" -body "012345\n"
}

# Start the server thread
server s1 -start

# Create a client thread called "c1"
client c1 {
    # Send a request
    txreq -url "/"
    # Wait for a response
    rxresp
# Insist that it be a success
expect resp.status == 200
}

# Run the client
client c1 -run

# Wait for the server to die
server s1 -wait

# (Forcefully) Stop the varnish instance.
varnish v1 -stop
```

### Example script output

The output, running this script looks as follows. The "bargraph" at the beginning of the line is an indication of the level of detail in the line. The second field where the message comes from. The rest of the line is anyones guess :-)

```
#  TEST tests/b00000.vtc starting
### v1  CMD: cd ../varnishd && ./varnishd -d -d -n v1 -a :9081 -T :9001 -b localhost:9080
### v1  opening CLI connection
#### v1  debug| NB: Storage size limited to 2GB on 32 bit architecture,\n
#### v1  debug| NB: otherwise we could run out of address space.\n
#### v1  debug| storage_file: filename: ./varnish.Shkoq5 (unlinked) size 2047 MB.\n
### v1  CLI connection fd = 3
#### v1  CLI TX| start
#### v1  debug| Using old SHMFILE\n
#### v1  debug| Notice: locking SHMFILE in core failed: Operation not permitted\n
#### v1  debug| bind(): Address already in use\n
#### v1  debug| rolling(1)...
#### v1  debug| \n
#### v1  debug| rolling(2)...\n
#### v1  debug| Debugging mode, enter "start" to start child\n
### v1  CLI 200 <start>
##   s1  Starting server
### s1  listen on :9080 (fd 6)
##   c1  Starting client
##   c1   Waiting for client
##   s1   started on :9080
##   c1   started
###  c1   connect to :9081
###  c1   connected to :9081 fd is 8
#### c1   | GET / HTTP/1.1\r\n
#### c1   | \r\n
###  c1   rxresp
#### s1   Accepted socket 7
###  s1   rxreq
#### s1   | GET / HTTP/1.1\r\n
#### s1   | X-Varnish: 422080121\r\n
#### s1   | X-Forwarded-For: 127.0.0.1\r\n
#### s1   | Host: localhost\r\n
#### s1   | \r\n
#### s1   http[ 0] | GET
#### s1   http[ 1] | /
#### s1   http[ 2] | HTTP/1.1
#### s1   http[ 3] | X-Varnish: 422080121
#### s1   http[ 4] | X-Forwarded-For: 127.0.0.1
#### s1   http[ 5] | Host: localhost
#### s1   | HTTP/1.1 200 Ok\r\n
#### s1   | Connection: close\r\n
#### s1   | \r\n
#### s1   | 012345\n
#### s1   | \r\n
##   s1   ending
#### c1   | HTTP/1.1 200 Ok\r\n
#### c1   | Content-Length: 9\r\n
#### c1   | Date: Mon, 16 Jun 2008 22:16:55 GMT\r\n
#### c1   | X-Varnish: 422080121\r\n
#### c1   | Age: 0\r\n
#### c1   | Via: 1.1 varnish\r\n
#### c1   | Connection: keep-alive\r\n
#### c1   | \r\n
#### c1   http[ 0] | HTTP/1.1
#### c1   http[ 1] | 200
#### c1   http[ 2] | Ok
#### c1   http[ 3] | Content-Length: 9
#### c1   http[ 4] | Date: Mon, 16 Jun 2008 22:16:55 GMT
#### c1   http[ 5] | X-Varnish: 422080121
#### c1   http[ 6] | Age: 0
#### c1   http[ 7] | Via: 1.1 varnish
#### c1   http[ 8] | Connection: keep-alive
#### c1   EXPECT resp.status (200) == 200 (200) match
##   c1   ending
##   s1   Waiting for server
#### v1   CLI TX| stop
###  v1   CLI 200 <stop>
#    TEST tests/b00000.vtc completed
```

If instead of 200 we had expected 201 with the line::

```
expect resp.status == 201
```

The output would have ended with::

```
#### c1   http[ 0] | HTTP/1.1
#### c1   http[ 1] | 200
```

```
#### c1  http[ 2] | Ok
#### c1  http[ 3] | Content-Length: 9
#### c1  http[ 4] | Date: Mon, 16 Jun 2008 22:26:35 GMT
#### c1  http[ 5] | X-Varnish: 648043653 648043652
#### c1  http[ 6] | Age: 6
#### c1  http[ 7] | Via: 1.1 varnish
#### c1  http[ 8] | Connection: keep-alive
---- c1  EXPECT resp.status (200) == 201 (201) failed
```

### SEE ALSO

- varnishhist(1)
- varnishlog(1)
- varnishncsa(1)
- varnishstat(1)
- varnishtop(1)
- vcl(7)

### HISTORY

The varnishtest program was developed by Poul-Henning Kamp (phk@phk.freebsd.dk) in cooperation with Varnish Software AS. This manual page was written by Stig Sandbeck Mathisen (ssm@linpro.no) using examples by Poul-Henning Kamp (phk@phk.freebsd.dk).

### COPYRIGHT

This document is licensed under the same licence as Varnish itself. See LICENCE for details.

- Copyright (c) 2007-2008 Varnish Software AS

## 3.12 varnishtop

### 3.12.1 Varnish log entry ranking

**Author** Dag-Erling Smørgrav

**Date** 2010-05-31

**Version** 1.0

**Manual section** 1

### SYNOPSIS

varnishtop [-1] [-b] [-C] [-c] [-d] [-f] [-I regex] [-i tag] [-n varnish_name] [-r file] [-V] [-X regex] [-x tag]

### DESCRIPTION

The varnishtop utility reads `varnishd(1)` shared memory logs and presents a continuously updated list of the most commonly occurring log entries. With suitable filtering using the `-I`, `-i`, `-X` and `-x` options, it can be used to display a ranking of requested documents, clients, user agents, or any other information which is recorded in the log.

The following options are available:

| | |
|---|---|
| **-1** | Instead of presenting of a continuously updated display, print the statistics once and exit. Implies `-d`. |
| **-b** | Include log entries which result from communication with a backend server. If neither `-b` nor `-c` is specified, varnishtop acts as if they both were. |
| **-C** | Ignore case when matching regular expressions. |
| **-c** | Include log entries which result from communication with a client. If neither `-b` nor `-c` is specified, varnishtop acts as if they both were. |
| **-d** | Process old log entries on startup. Normally, varnishtop will only process entries which are written to the log after it starts. |
| **-f** | Sort and group only on the first field of each log entry. This is useful when displaying e.g. stataddr entries, where the first field is the client IP address. |
| **-I regex** | Include log entries which match the specified regular expression. If neither `-I` nor `-i` is specified, all log entries are included. |
| **-i tag** | Include log entries with the specified tag. If neither `-I` nor `-i` is specified, all log entries are included. |
| **-p period** | Specifies the number of seconds to measure over, the default is 60 seconds. The first number in the list is the average number of requests seen over this time period. |
| **-n** | Specifies the name of the varnishd instance to get logs from. If `-n` is not specified, the host name is used. |
| **-r file** | Read log entries from file instead of shared memory. |
| **-V** | Display the version number and exit. |
| **-X regex** | Exclude log entries which match the specified regular expression. |
| **-x tag** | Exclude log entries with the specified tag. |

### EXAMPLES

The following example displays a continuously updated list of the most frequently requested URLs::

```
varnishtop -i RxURL
```

The following example displays a continuously updated list of the most commonly used user agents::

```
varnishtop -i RxHeader -C -I ^User-Agent
```

### SEE ALSO

- varnishd(1)
- varnishhist(1)

- varnishlog(1)
- varnishncsa(1)
- varnishstat(1)

### HISTORY

The varnishtop utility was originally developed by Poul-Henning Kamp in cooperation with Verdens Gang AS and Varnish Software AS, and later substantially rewritten by Dag-Erling Smørgrav. This manual page was written by Dag-Erling Smørgrav.

### COPYRIGHT

## 3.13 Shared Memory Logging and Statistics

Varnish uses shared memory for logging and statistics, because it is faster and much more efficient. But it is also tricky in ways a regular logfile is not.

When you open a file in "append" mode, the operating system guarantees that whatever you write will not overwrite existing data in the file. The neat result of this is that multiple procesess or threads writing to the same file does not even need to know about each other, it all works just as you would expect.

With a shared memory log, we get no help from the kernel, the writers need to make sure they do not stomp on each other, and they need to make it possible and safe for the readers to access the data.

The "CS101" way, is to introduce locks, and much time is spent examining the relative merits of the many kinds of locks available.

Inside the varnishd (worker) process, we use mutexes to guarantee consistency, both with respect to allocations, log entries and stats counters.

We do not want a varnishncsa trying to push data through a stalled ssh connection to stall the delivery of content, so readers like that are purely read-only, they do not get to affect the varnishd process and that means no locks for them.

Instead we use "stable storage" concepts, to make sure the view seen by the readers is consistent at all times.

As long as you only add stuff, that is trivial, but taking away stuff, such as when a backend is taken out of the configuration, we need to give the readers a chance to discover this, a "cooling off" period.

When Varnishd starts, if it finds an existing shared memory file, and it can safely read the master_pid field, it will check if that process is running, and if so, fail with an error message, indicating that -n arguments collide.

In all other cases, it will delete and create a new shmlog file, in order to provide running readers a cooling off period, where they can discover that there is a new shmlog file, by doing a stat(2) call and checking the st_dev & st_inode fields.

### 3.13.1 Allocations

Sections inside the shared memory file are allocated dynamically, for instance when a new backend is added.

While changes happen to the linked list of allocations, the "alloc_seq" header field is zero, and after the change, it gets a value different from what it had before.

### 3.13.2 Deallocations

When a section is freed, its class will change to "Cool" for at least 10 seconds, giving programs using it time to detect the change in alloc_seq header field and/or the change of class.

## 3.14 VMOD - Varnish Modules

For all you can do in VCL, there are things you can not do. Look an IP number up in a database file for instance. VCL provides for inline C code, and there you can do everything, but it is not a convenient or even readable way to solve such problems.

This is where VMODs come into the picture: A VMOD is a shared library with some C functions which can be called from VCL code.

For instance:

```
import std;

sub vcl_deliver {
        set resp.http.foo = std.toupper(req.url);
}
```

The "std" vmod is one you get with Varnish, it will always be there and we will put "boutique" functions in it, such as the "toupper" function shown above. The full contents of the "std" module is documented in XXX:TBW.

This part of the manual is about how you go about writing your own VMOD, how the language interface between C and VCC works etc. This explanation will use the "std" VMOD as example, having a varnish source tree handy may be a good idea.

### 3.14.1 The vmod.vcc file

The interface between your VMOD and the VCL compiler ("VCC") and the VCL runtime ("VRT") is defined in the vmod.vcc file which a python script called "vmod.py" turns into thaumathurgically challenged C data structures that does all the hard work.

The std VMODs vmod.vcc file looks somewhat like this:

```
Module std
Init init_function
Function STRING toupper(PRIV_CALL, STRING_LIST)
Function STRING tolower(PRIV_VCL, STRING_LIST)
Function VOID set_ip_tos(INT)
```

The first line gives the name of the module, nothing special there.

The second line specifies an optional "Init" function, which will be called whenever a VCL program which imports this VMOD is loaded. This gives a chance to initialize the module before any of the functions it implements are called.

The next three lines specify two functions in the VMOD, along with the types of the arguments, and that is probably where the hardest bit of writing a VMOD is to be found, so we will talk about that at length in a moment.

Notice that the third function returns VOID, that makes it a "procedure" in VCL lingo, meaning that it cannot be used in expressions, right side of assignments and such places. Instead it can be used as a primary action, something functions which return a value can not:

```
sub vcl_recv {
        std.set_ip_tos(32);
}
```

Running vmod.py on the vmod.vcc file, produces an "vcc_if.c" and "vcc_if.h" files, which you must use to build your shared library file.

Forget about vcc_if.c everywhere but your Makefile, you will never need to care about its contents, and you should certainly never modify it, that voids your warranty instantly.

But vcc_if.h is important for you, it contains the prototypes for the functions you want to export to VCL.

For the std VMOD, the compiled vcc_if.h file looks like this:

```
struct sess;
struct VCL_conf;
const char * vmod_toupper(struct sess *, struct vmod_priv *, const char *, ...);
const char * vmod_tolower(struct sess *, struct vmod_priv *, const char *, ...);
int meta_function(void **, const struct VCL_conf *);
```

Those are your C prototypes. Notice the `vmod_` prefix on the function names and the C-types as return types and arguments.

## 3.14.2 VCL and C data types

VCL data types are targeted at the job, so for instance, we have data types like "DURATION" and "HEADER", but they all have some kind of C language representation. Here is a description of them, from simple to nasty.

**INT** C-type: `int`

An integer as we know and love them.

**REAL** C-type: `double`

A floating point value

**DURATION** C-type: `double`

Units: seconds

A time interval, as in "25 minutes".

**TIME** C-type: `double`

Units: seconds since UNIX epoch

An absolute time, as in "Mon Sep 13 19:06:01 UTC 2010".

**STRING** C-type: `const char *`

A NUL-terminated text-string.

Can be NULL to indicate that the nonexistent string, for instance:

```
mymod.foo(req.http.foobar);
```

If there were no "foobar" HTTP header, the vmod_foo() function would be passed a NULL pointer as argument.

When used as a return value, the producing function is responsible for arranging memory management. Either by freeing the string later by whatever means available or by using storage allocated from the session or worker workspaces.

**STRING_LIST** C-type: `const char *, ...`

A multi-component text-string. We try very hard to avoid doing text-processing in Varnish, and this is one way we do that, by not editing separate pieces of a sting together to one string, until we need to.

Consider this contrived example:

```
set bereq.http.foo = std.toupper(req.http.foo + req.http.bar);
```

The usual way to do this, would be be to allocate memory for the concatenated string, then pass that to `toupper()` which in turn would return another freshly allocated string with the modified result. Remember: strings in VCL are `const`, we cannot just modify the string in place.

What we do instead, is declare that `toupper()` takes a "STRING_LIST" as argument. This makes the C function implementing `toupper()` a vararg function (see the prototype above) and responsible for considering all the `const char *` arguments it finds, until the magic marker "vrt_magic_string_end" is encountered.

Bear in mind that the individual strings in a STRING_LIST can be NULL, as described under STRING, that is why we do not use NULL as the terminator.

Right now we only support STRING_LIST being the last argument to a function, we may relax that at a latter time.

If you don't want to bother with STRING_LIST, just use STRING and make sure your sess_workspace param is big enough.

**PRIV_VCL** See below

**PRIV_CALL** See below

**VOID** C-type: `void`

Can only be used for return-value, which makes the function a VCL procedure.

**IP, BOOL, HEADER** XXX: these types are not released for use in vmods yet.

### 3.14.3 Private Pointers

It is often useful for library functions to maintain local state, this can be anything from a precompiled regexp to open file descriptors and vast data structures.

The VCL compiler supports two levels of private pointers, "per call" and "per VCL"

"per call" private pointers are useful to cache/store state relative to the specific call or its arguments, for instance a compiled regular expression specific to a regsub() statement or a simply caching the last output of some expensive lookup.

"per vcl" private pointers are useful for such global state that applies to all calls in this VCL, for instance flags that determine if regular expressions are case-sensitive in this vmod or similar.

The way it works in the vmod code, is that a `struct vmod_priv *` is passed to the functions where argument type PRIV_VCL or PRIV_CALL is specified.

This structure contains two members:

```
typedef void vmod_priv_free_f(void *);
struct vmod_priv {
        void                    *priv;
        vmod_priv_free_f        *free;
};
```

The “priv” element can be used for whatever the vmod code wants to use it for, it defaults to a NULL pointer.

The “free” element defaults to NULL, and it is the modules responsibility to set it to a suitable function, which can clean up whatever the “priv” pointer points to.

When a VCL program is discarded, all private pointers are checked to see if both the “priv” and “free” elements are non-NULL, and if they are, the “free” function will be called with the “priv” pointer as only argument. The “per vcl” pointers is guaranteed to be the last one inspected.

## 3.15 vmod_std

### 3.15.1 Varnish Standard Module

**Author** Per Buer

**Date** 2011-05-19

**Version** 1.0

**Manual section** 3

#### SYNOPSIS

import std

#### DESCRIPTION

The Varnish standard module contains useful, generic function that don’t quite fit in the VCL core, but are still considered very useful to a broad audience.

#### FUNCTIONS

##### toupper

**Prototype** toupper(STRING S)

**Return value** String

**Description** Converts the STRING S to upper case.

**Example** set beresp.http.x-scream = std.toupper(“yes!”);

##### tolower

**Prototype** tolower(STRING S)

**Return value** String

**Description** Converts the STRING to lower case.

**Example** set beresp.http.x-nice = std.tolower("VerY");

### set_up_tos

**Prototype** set_ip_tos(INT I)

**Return value** Void

**Description** Sets the Type-of-Service flag for the current session. Please note that the TOS flag is not removed by the end of the request so probably want to set it on every request should you utilize it.

**Example**

```
if (req.url ~ ^/slow/) {
        std.set_up_tos(0x0);
}
```

### random

**Prototype** random(REAL a, REAL b)

**Return value** Real

**Description** Returns a random REAL number between *a* and *b*.

**Example** set beresp.http.x-random-number = std.random(1, 100);

### log

**Prototype** log(STRING string)

**Return value** Void

**Description** Logs string to the shared memory log.

**Example** std.log("Something fishy is going on with the vhost " + req.host);

### syslog

**Prototype** syslog(INT priority, STRING string)

**Return value** Void

**Description** Logs *string* to syslog marked with *priority*.

**Example** std.syslog( LOG_USER|LOG_ALERT, "There is serious troble");

### fileread

**Prototype** fileread(STRING filename)

**Return value** String

**Description** Reads a file and returns a string with the content. Please note that it is not recommended to send variables to this function the caching in the function doesn't take this into account. Also, files are not re-read.

**Example** set beresp.http.x-served-by = std.fileread("/etc/hostname");

#### duration

**Prototype** duration(STRING s, DURATION fallback)

**Return value** Duration

**Description** Converts the string s to seconds. s can be quantified with the usual s (seconds), m (minutes), h (hours), d (days) and w (weeks) units. If it fails to parse the string *fallback* will be used

**Example** set beresp.ttl = std.duration("1w", 3600);

#### integer

**Prototype** integer(STRING s, INT fallback)

**Return value** Int

**Description** Converts the string s to an integer. If it fails to parse the string *fallback* will be used

**Example** if (std.integer(beresp.http.x-foo, 0) > 5) { ... }

#### collect

**Prototype** collect(HEADER header)

**Return value** Void

**Description** Collapses the header, joining the headers into one.

**Example** std.collect(req.http.cookie); This will collapse several Cookie: headers into one, long cookie header.

### SEE ALSO

- vcl(7)
- varnishd(1)

### HISTORY

The Varnish standard module was released along with Varnish Cache 3.0. This manual page was written by Per Buer with help from Martin Blix Grydeland.

### COPYRIGHT

This document is licensed under the same licence as Varnish itself. See LICENCE for details.

- Copyright (c) 2011 Varnish Software

## 3.16 Shared Memory Logging

### 3.16.1 TTL records

A TTL record is emitted whenever the ttl, grace or keep values for an object is set.

The format is:

```
%u %s %d %d %d %d %d [ %d %u %u ]
|  |  |  |  |  |  |     |  |  |
|  |  |  |  |  |  |     |  |  +- Max-Age from Cache-Control header
|  |  |  |  |  |  |     |  +---- Expires header
|  |  |  |  |  |  |     +------- Date header
|  |  |  |  |  |  +------------ Age (incl Age: header value)
|  |  |  |  |  +--------------- Reference time for TTL
|  |  |  |  +------------------ Keep
|  |  |  +--------------------- Grace
|  |  +------------------------ TTL
|  +--------------------------- "RFC" or "VCL"
+------------------------------ object XID
```

The last three fields are only present in “RFC” headers.

Examples:

```
1001 RFC 19 -1 -1 1312966109 4 0 0 23
1001 VCL 10 -1 -1 1312966109 4
1001 VCL 7 -1 -1 1312966111 6
1001 VCL 7 120 -1 1312966111 6
1001 VCL 7 120 3600 1312966111 6
1001 VCL 12 120 3600 1312966113 8
```

### 3.16.2 Gzip records

A Gzip record is emitted for each instance of gzip or gunzip work performed. Worst case, an ESI transaction stored in gzip’ed objects but delivered gunziped, will run into many of these.

The format is:

```
%c %c %c %d %d %d %d %d
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  +- Bit length of compressed data
|  |  |  |  |  |  +---- Bit location of 'last' bit
|  |  |  |  |  +------- Bit location of first deflate block
|  |  |  |  +---------- Bytes output
|  |  |  +------------- Bytes input
|  |  +---------------- 'E' = ESI, '-' = Plain object
|  +------------------- 'F' = Fetch, 'D' = Deliver
+---------------------- 'G' = Gzip, 'U' = Gunzip, 'u' = Gunzip-test
```

Examples:

```
U F E 182 159 80 80 1392
G F E 159 173 80 1304 1314
```

CHAPTER

FOUR

# FREQUENTLY ASKED QUESTIONS

The most frequently asked questions about Varnish in different contexts.

## 4.1 General questions

### 4.1.1 What is Varnish?

Varnish is a state-of-the-art, high-performance web accelerator. It uses the advanced features in Linux 2.6, FreeBSD 6/7 and Solaris 10 to achieve its high performance.

Some of the features include

- A modern design
- VCL - a very flexible configuration language
- Load balancing with health checking of backends
- Partial support for ESI
- URL rewriting
- Graceful handling of “dead” backends

Features to come (Experimental):

- Support for Ranged headers
- Support for persistent cache

Varnish is free software and is licenced under a modified BSD licence. Please read the introduction to get started with Varnish.

### 4.1.2 How...

**How much RAM/Disk do I need for Varnish?**

That depends on pretty much everything.

I think our best current guidance is that you go for a cost-effective RAM configuration, something like 1-16GB, and an SSD.

Unless you positively know that you will need it, there is little point in spending a fortune on a hand-sewn motherboard that can fit several TB in special RAM blocks, riveted together by leftover watch-makers in Switzerland.

On the other hand, if you plot your traffic in Gb/s, you probably need all the RAM you can afford/get.

**How can I limit Varnish to use less RAM?**

You can not. Varnish operates in Virtual Memory and it is up to the kernel to decide which process gets to use how much RAM to map the virtual address-space of the process.

**How do I instruct varnish to ignore the query parameters and only cache one instance of an object?**

This can be achieved by removing the query parameters using a regexp:

```
sub vcl_recv {
    set req.url = regsub(req.url, "\?.*", "");
}
```

**How can I force a refresh on a object cached by varnish?**

Refreshing is often called purging a document. You can purge at least 2 different ways in Varnish:

1. Command line

   From the command line you can write:

   ```
   ban.url ^/$
   ```

   to ban your / document. As you might see ban.url takes an regular expression as its argument. Hence the `^` and `$` at the front and end. If the `^` is omitted, all the documents ending in a `/` in the cache would be deleted.

   So to delete all the documents in the cache, write:

   ```
   ban.url .
   ```

   at the command line.

2. HTTP PURGE

   VCL code to allow HTTP PURGE is to be found here. Note that this method does not support wildcard purging.

**How can I debug the requests of a single client?**

The "varnishlog" utility may produce a horrendous amount of output. To be able debug our own traffic can be useful.

The ReqStart token will include the client IP address. To see log entries matching this, type:

```
$ varnishlog -c -m ReqStart:192.0.2.123
```

To see the backend requests generated by a client IP address, we can match on the TxHeader token, since the IP address of the client is included in the X-Forwarded-For header in the request sent to the backend.

At the shell command line, type:

```
$ varnishlog -b -m TxHeader:192.0.2.123
```

**How can I rewrite URLS before they are sent to the backend?**

You can use the `regsub()` function to do this. Here's an example for zope, to rewrite URLs for the virtualhostmonster:

```
if (req.http.host ~ "^(www.)?example.com") {
  set req.url = regsub(req.url, "^", "/VirtualHostBase/http/example.com:80/Sites/example.com/VirtualH
}
```

**I have a site with many host names, how do I keep them from multiplying the cache?**

You can do this by normalizing the `Host` header for all your host names. Here's a VCL example:

```
if (req.http.host ~ "^(www.)?example.com") {
  set req.http.host = "example.com";
}
```

**How do I do to alter the request going to the backend?** You can use the `bereq` object for altering requests going to the backend, but you can only 'set' values to it. Therefore use req.url to build the request:

```
sub vcl_miss {
        set bereq.url = regsub(req.url,"stream/","/");
        return(fetch);
}
```

**How do I force the backend to send Vary headers?**

We have anecdotal evidence of non-RFC2616 compliant backends, which support content negotiation, but which do not emit a Vary header, unless the request contains Accept headers.

It may be appropriate to send no-op Accept headers to trick the backend into sending us the Vary header.

The following should be sufficient for most cases:

```
Accept: */*
Accept-Language: *
Accept-Charset: *
Accept-Encoding: identity
```

Note that Accept-Encoding can not be set to `*`, as the backend might then send back a compressed response which the client would be unable to process.

This can of course be implemented in VCL.

**How can I customize the error messages that Varnish returns?**

A custom error page can be generated by adding a `vcl_error` to your configuration file. The default error page looks like this:

```
sub vcl_error {
    set obj.http.Content-Type = "text/html; charset=utf-8";

    synthetic {"
    <?xml version="1.0" encoding="utf-8"?>
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
    <html>
      <head>
        <title>"} + obj.status + " " + obj.response + {"</title>
      </head>
      <body>
      <h1>Error "} + obj.status + " " + obj.response + {"</h1>
      <p>"} + obj.response + {"</p>
        <h3>Guru Meditation:</h3>
        <p>XID: "} + req.xid + {"</p>
        <address><a href="http://www.varnish-cache.org/">Varnish</a></address>
      </body>
     </html>
     "};
    return(deliver);
}
```

**How do I instruct varnish to ignore the query parameters and only cache one instance of an object?**

This can be achieved by removing the query parameters using a regexp:

```
sub vcl_recv {
    set req.url = regsub(req.url, "\?.*", "");
}
```

### 4.1.3 Where...

**Can I find varnish for my operating system?**

We know that Varnish has been packaged for Debian, Ubuntu, RHEL, CentOS, (Open)SUSE, Gentoo and FreeBSD, possibly more. Check whatever package manager you use. Or read *Installing Varnish on your computer*.

### 4.1.4 Can I...

**Can I use Varnish as a client-side proxy?**

No. Varnish needs all backends configured in the VCL. Look at squid instead.

**Can I run Varnish on a 32bit system?**

Yes, recently somebody even claimed to run Varnish on his N900 mobile phone recently, but if you have the choice, go 64 bit from the start.

Varnish is written to use Virtual Memory and on a 32bit system that really cramps your style, and you will have trouble configuring more than 2 GB of storage.

**Can I run Varnish on the same system as Apache?**

Yes, and many people do that with good success.

There will be competition for resources, but Apache is not particular good at using RAM effectively and Varnish is, so this synergy usually more than compensates for the competition.

**Can I run multiple Varnish on the same system?**

Yes, as long as you give them different TCP ports and different `-n` arguments, you will be fine.

**Can I cache multiple virtual hosts with one Varnish?**

Yes, that works right out of the box.

**Can I see what is cached in Varnish?**

That is not possible for several reasons. A command to list all the contents of a Varnish cache with millions of objects would bring your Varnish to a standstill while it traverses the index.

Besides, the output is a lot less useful than you might think.

**Can I use Varnish to do HTTPS?**

Not at present, and while we keep an eye on this, there are no current plans to add HTTPS support, until we can find a way where it adds significant value, relative to running a stand-alone HTTPS proxy such as nginx or pound.

**Can Varnish load balance between multiple backends?**

Yes, you need VCL code like this:

```
director foobar round-robin {
    { .backend = { .host = "www1.example.com"; .port = "http"; } }
    { .backend = { .host = "www2.example.com"; .port = "http"; } }
}

sub vcl_recv {
```

```
            set req.backend = foobar;
    }

(XXX: reference to docs, once written)
```

### 4.1.5 Why ...

**Why does it look like Varnish sends all requests to the backend? I thought it was a cache?**

**There are 2 common reasons for this:**

1. The object's `ttl expired`. A common situation is that the backend does not set an expiry time on the requested image/file/webpage, so Varnish uses the default TTL (normally 120s).
2. **Your site uses `cookies`:**
   - By default, varnish will not cache `responses` from the backend that come with a `Set-Cookie`: header.
   - By default, varnish will not serve `requests` with a `Cookie:` header, but pass them to the backend instead. Check out [wiki:VCLExamples these VCL examples] on how to make varnish cache cookied/logged in users sanely.

**Why are regular expressions case-sensitive?**

Some HTTP headers, such as `Host:` and `Location:` contain fully qualified domain names, which by definition is not case-sensitive. Other HTTP headers are case-sensitive, most notably the URLs. Therefore a "one size fits all" solution is not possible.

In previous releases, we used the POSIX regular expressions supplied with the operating system, and decided, because the most common use of regexps were on **`**`Host:`**`** headers, that they should not be case-sensitive.

From version 2.1.0 and forward, we use PCRE regular expressions, where it *is* possible to control case-sensitivity in the individual regular expressions, so we decided that it would probably confuse people if we made the default case-insensitive. (We promise not to change our minds about this again.)

To make a PCRE regex case insensitive, put `(?i)` at the start:

```
if (req.http.host ~ "(?i)example.com$") {
        ...
}
```

See the PCRE man pages for more information.

**Are regular expressions case sensitive or not? Can I change it?**

In 2.1 and newer, regular expressions are case sensitive by default. In earlier versions, they were case insensitive.

To change this for a single regex in 2.1, use `(?i)` at the start.

See the PCRE man pages for more information.

**Why does the ‘‘Via:‘‘ header say 1.1 in Varnish 2.1.x?**

The number in the `Via:` header is the HTTP protocol version supported/applied, not the software's version number.

**Why did you call it *Varnish*?**

Long story, but basically the instigator of Varnish spent a long time staring at an art-poster with the word "Vernisage" and ended up checking it in a dictionary, which gives the following three meanings of the word:

r.v. var·nished, var·nish·ing, var·nish·es

1. To cover with varnish.
2. To give a smooth and glossy finish to.
3. To give a deceptively attractive appearance to; gloss over.

The three point describes happens to your backend system when you put Varnish in front of it.

**Why does Varnish require the system to have a C compiler?**

The *VCL* compiler generates C source as output (your config file), and uses the systems C-compiler to compile that into a shared library. If there is no C compiler, Varnish will not work.

**Isn't that security problem?**

The days when you could prevent people from running non-approved programs by removing the C compiler from your system ended roughly with the VAX 11/780 computer.

### 4.1.6 Troubleshooting

**Why am I getting a cache hit, but a request is still going to my backend?**

Varnish has a feature called **hit for pass**, which is used when Varnish gets a response from the backend and finds out it cannot be cached. In such cases, Varnish will create a cache object that records that fact, so that the next request goes directly to "pass".

**Since Varnish bundles multiple requests for the same URL to the backend, a common case where a client will get a hit for**

- Client 1 requests url /foo
- Client 2..N request url /foo
- Varnish tasks a worker to fetch /foo for Client 1
- Client 2..N are now queued pending response from the worker
- Worker returns object to varnish which turns out to be non-cacheable.
- Client 2..N are now given the **hit for pass** object instructing them to go to the backend

The **hit for pass** object will stay cached for the duration of its ttl. This means that subsequent clients requesting /foo will be sent straight to the backend as long as the **hit for pass** object exists. The **varnishstat** can tell you how many **hit for pass** objects varnish has served. The default vcl will set ttl for a hit_for_pass object to 120s. But you can override this, using the following logic:

```
sub vcl_fetch {
  if (!obj.cacheable) {
    # Limit the lifetime of all 'hit for pass' objects to 10 seconds
    obj.ttl = 10s;
    return(hit_for_pass);
  }
}
```

## 4.2 HTTP

**What is the purpose of the X-Varnish HTTP header?**

The X-Varnish HTTP header allows you to find the correct log-entries for the transaction. For a cache hit, X-Varnish will contain both the ID of the current request and the ID of the request that populated the cache. It makes debugging Varnish a lot easier.

**Does Varnish support compression?**

This is a simple question with a complicated answer; see WIKI.

**How do I add a HTTP header?**

To add a HTTP header, unless you want to add something about the client/request, it is best done in vcl_fetch as this means it will only be processed every time the object is fetched:

```
sub vcl_fetch {
  # Add a unique header containing the cache servers IP address:
  remove beresp.http.X-Varnish-IP;
  set    beresp.http.X-Varnish-IP = server.ip;
  # Another header:
  set    beresp.http.Foo = "bar";
}
```

**How can I log the client IP address on the backend?**

All I see is the IP address of the varnish server. How can I log the client IP address?

We will need to add the IP address to a header used for the backend request, and configure the backend to log the content of this header instead of the address of the connecting client (which is the varnish server).

Varnish configuration:

```
sub vcl_recv {
  # Add a unique header containing the client address
  remove req.http.X-Forwarded-For;
  set    req.http.X-Forwarded-For = client.ip;
  # [...]
}
```

For the apache configuration, we copy the “combined” log format to a new one we call “varnishcombined”, for instance, and change the client IP field to use the content of the variable we set in the varnish configuration:

```
LogFormat "%{X-Forwarded-For}i %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" varnishcomb
```

And so, in our virtualhost, you need to specify this format instead of “combined” (or “common”, or whatever else you use):

```
<VirtualHost *:80>
  ServerName www.example.com
  # [...]
  CustomLog /var/log/apache2/www.example.com/access.log varnishcombined
  # [...]
</VirtualHost>
```

The [http://www.openinfo.co.uk/apache/index.html mod_extract_forwarded Apache module] might also be useful.

## 4.3 Configuration

### 4.3.1 VCL

**What is VCL?**

VCL is an acronym for Varnish Configuration Language. In a VCL file, you configure how Varnish should behave. Sample VCL files will be included in this Wiki at a later stage.

**Where is the documentation on VCL?**

Please see `man 7 vcl`. There are also some real-world examples on the wiki

**How do I load VCL file while Varnish is running?**

- Place the VCL file on the server
- Telnet into the managment port.
- do a "vcl.load <configname> <filename>" in managment interface. <configname> is whatever you would like to call your new configuration.
- do a "vcl.use <configname>" to start using your new config.

**Should I use ''pipe" or ''pass" in my VCL code? What is the difference?**

When varnish does a `pass` it acts like a normal HTTP proxy. It reads the request and pushes it onto the backend. The next HTTP request can then be handled like any other.

`pipe` is only used when Varnish for some reason can't handle the `pass`. `pipe` reads the request, pushes in onty the backend _only_ pushes bytes back and forth, with no other actions taken.

Since most HTTP clients do pipeline several requests into one connection this might give you an undesirable result - as every subsequent request will reuse the existing `pipe`. Please see *this article <http://www.varnish-cache.org/trac/wiki/VCLExamplePipe>* for more details and a workaround.

Varnish versions prior to 2.0 does not support handling a request body with `pass` mode, so in those releases `pipe` is required for correct handling.

In 2.0 and later, `pass` will handle the request body correctly.

If you get 503 errors when making a request which is `pass` ed, make sure that you're specifying the backend before returning from vcl_recv with `pass`.

## 4.4 Logging

**Where can I find the log files?**

Varnish does not log to a file, but to shared memory log. Use the varnishlog utility to print the shared memory log or varnishncsa to present it in the Apache / NCSA "combined" log format.

CHAPTER

FIVE

# POUL-HENNINGS RANDOM OUTBURSTS

You may or may not want to know what Poul-Henning think.

## 5.1 Thoughts on the eve of Varnish 3.0

Five years ago, I was busy transforming my pile of random doddles on 5mm squared paper into software, according to "git log" working on the first stevedores.

In two weeks I will be attending the Varnish 3.0 release party in Oslo.

Sometimes I feel that development of Varnish takes for ever and ever, and that it must be like watching paint dry for the users, but 3 major releases in 5 years is actually not too shabby come to think of it.

Varnish 3.0 "only" has two big new features, VMOD and GZIP, and a host of smaller changes, which you will notice if they are new features, and not notice if they are bug fixes.

GZIP will probably be most important to the ESI users, and I wonder if all the time I spent fiddling bits in the middle of compressed data pays off, or if the concept of patchwork-quilting GZIP files was a bad idea from end to other.

VMODs on the other hand, was an instant success, because they make it much easier for people to extend Varnish with new functionality, and I know of several VMODs in the pipeline which will make it possible to do all sorts of wonderful things from VCL.

All in all, I feel happy about the 3.0 release, and I hope the users will too.

We are not finished of course, ideas and patches for Varnish 4.0 are already starting to pile up, and hopefully we can get that into a sensible shape 18 months from now, late 2012-ish.

> "Life is what happens to you while you're busy making other plans"

said John Lennon, a famous murder victim from New York.

I feel a similar irony in the way Varnish happened to me:

My homepage is written in raw HTML using the vi(1) editor, runs on a book-sized Soekris NET5501 computer, averages 50 hits a day with an Alexa rank just north of the 3.5 million mark. A normal server with Varnish could deliver all traffic my webserver has ever delivered, in less than a second.

But varnish-cache.org has Alexa rank around 30.000, "varnish cache" shows a nice trend on Google and #varnish confuses the heck out of teenage girls and wood workers on Twitter, so clearly I am doing something right.

I still worry about the The Fraud Police though, "I have no idea what I'm doing, and I totally make shit up as I go along." is a disturbingly precise summary of how I feel about my work in Varnish.

The Varnish 3.0 release is therefore dedicated to all the kind Varnish developers and users, who have tested, reported bugs, suggested ideas and generally put up with me and my bumbling ways for these past five years.

Much appreciated,

Poul-Henning, 2011-06-02

## 5.2 Why no SSL ?

This is turning into a bit of a FAQ, but the answer is too big to fit in the margin we use for those.

There are a number of reasons why there are no plans in sight that will grow SSL support in Varnish.

First, I have yet to see a SSL library where the source code is not a nightmare.

As I am writing this, the varnish source-code tree contains 82.595 lines of .c and .h files, including JEmalloc (12.236 lines) and Zlib (12.344 lines).

OpenSSL, as imported into FreeBSD, is 340.722 lines of code, nine times larger than the Varnish source code, 27 times larger than each of Zlib or JEmalloc.

This should give you some indication of how insanely complex the canonical implementation of SSL is.

Second, it is not exactly the best source-code in the world. Even if I have no idea what it does, there are many aspect of it that scares me.

Take this example in a comment, randomly found in s3-srvr.c:

```
/* Throw away what we have done so far in the current handshake,
 * which will now be aborted. (A full SSL_clear would be too much.)
 * I hope that tmp.dh is the only thing that may need to be cleared
 * when a handshake is not completed ... */
```

I hope they know what they are doing, but this comment doesn't exactly carry that point home, does it ?

But let us assume that a good SSL library can be found, what would Varnish do with it ?

We would terminate SSL sessions, and we would burn CPU cycles doing that. You can kiss the highly optimized delivery path in Varnish goodby for SSL, we cannot simply tell the kernel to put the bytes on the socket, rather, we have to corkscrew the data through the SSL library and then write it to the socket.

Will that be significantly different, performance wise, from running a SSL proxy in separate process ?

No, it will not, because the way varnish would have to do it would be to ... start a separate process to do the SSL handling.

There is no other way we can guarantee that secret krypto-bits do not leak anywhere they should not, than by fencing in the code that deals with them in a child process, so the bulk of varnish never gets anywhere near the certificates, not even during a core-dump.

Would I be able to write a better stand-alone SSL proxy process than the many which already exists ?

Probably not, unless I also write my own SSL implementation library, including support for hardware crypto engines and the works.

That is not one of the things I dreamt about doing as a kid and if I dream about it now I call it a nightmare.

So the balance sheet, as far as I can see it, lists "It would be a bit easier to configure" on the plus side, and everything else piles up on the minus side, making it a huge waste of time and effort to even think about it..

Poul-Henning, 2011-02-15

## 5.3 How GZIP, and GZIP+ESI works in Varnish

First of all, everything you read about GZIP here, is controlled by the parameter:

> http_gzip_support

Which defaults to "on" if you do not want Varnish to try to be smart about compression, set it to "off" instead.

### 5.3.1 What does http_gzip_support do

A request which is sent into 'pipe' or 'pass' mode from vcl_recv{} will not experience any difference, this processing only affects cache hit/miss requests.

Unless vcl_recv{} results in "pipe" or "pass", we determine if the client is capable of receiving gzip'ed content. The test amounts to:

> Is there a Accept-Encoding header that mentions gzip, and if is has a q=# number, is it larger than zero.

Clients which can do gzip, gets their header rewritten to:

> Accept-Encoding: gzip

And clients which do not support gzip gets their Accept-Encoding header removed. This ensures conformity with respect to creating Vary: strings during object creation.

During lookup, we ignore any "Accept-encoding" in objects Vary: strings, to avoid having a gzip and gunzip'ed version of the object, varnish can gunzip on demand. (We implement this bit of magic at lookup time, so that any objects stored in persistent storage can be used with or without gzip support enabled.)

Varnish will not do any other types of compressions than gzip, in particular we will not do deflate, as there are browser bugs in that case.

Before vcl_miss{} is called, the backend requests Accept-Encoding is always set to:

> Accept-Encoding: gzip

Even if this particular client does not support

To always entice the backend into sending us gzip'ed content.

Varnish will not gzip any content on its own (but see below), we trust the backend to know what content can be sensibly gzip'ed (html) and what can not (jpeg)

If in vcl_fetch{} we find out that we are trying to deliver a gzip'ed object to a client that has not indicated willingness to receive gzip, we will ungzip the object during deliver.

### 5.3.2 Tuning, tweaking and frobbing

In vcl_recv{} you have a chance to modify the client's Accept-Encoding: header before anything else happens.

In vcl_pass{} the clients Accept-Encoding header is copied to the backend request unchanged. Even if the client does not support gzip, you can force the A-C header to "gzip" to save bandwidth between the backend and varnish, varnish will gunzip the object before delivering to the client.

In vcl_miss{} you can remove the "Accept-Encoding: gzip" header, if you do not want the backend to gzip this object.

In vcl_fetch{} two new variables allow you to modify the gzip-ness of objects during fetch:

> set beresp.do_gunzip = true;

Will make varnish gunzip an already gzip'ed object from the backend during fetch. (I have no idea why/when you would use this...)

> set beresp.do_gzip = true;

Will make varnish gzip the object during fetch from the backend, provided the backend didn't send us a gzip'ed object.

Remember that a lot of content types cannot sensibly be gziped, most notably compressed image formats like jpeg, png and similar, so a typical use would be:

```
sub vcl_fetch {
        if (req.url ~ "html$") {
                set beresp.do_gzip = true;
        }
}
```

### 5.3.3 GZIP and ESI

First, note the new syntax for activating ESI:

```
sub vcl_fetch {
        set beresp.do_esi = true;
}
```

In theory, and hopefully in practice, all you read above should apply also when you enable ESI, if not it is a bug you should report.

But things are vastly more complicated now. What happens for instance, when the backend sends a gzip'ed object we ESI process it and it includes another object which is not gzip'ed, and we want to send the result gziped to the client ?

Things can get really hairy here, so let me explain it in stages.

Assume we have a ungzipped object we want to ESI process.

The ESI parser will run through the object looking for the various magic strings and produce a byte-stream we call the "VEC" for Varnish ESI Codes.

The VEC contains instructions like "skip 234 bytes", "deliver 12919 bytes", "include /foobar", "deliver 122 bytes" etc and it is stored with the object.

When we deliver an object, and it has a VEC, special esi-delivery code interprets the VEC string and sends the output to the client as ordered.

When the VEC says "include /foobar" we do what amounts to a restart with the new URL and possibly Host: header, and call vcl_recv{} etc. You can tell that you are in an ESI include by examining the 'req.esi_level' variable in VCL.

The ESI-parsed object is stored gzip'ed under the same conditions as above: If the backend sends gzip'ed and VCL did not ask for do_gunzip, or if the backend sends ungzip'ed and VCL asked for do_gzip.

Please note that since we need to insert flush and reset points in the gzip file, it will be slightly larger than a normal gzip file of the same object.

When we encounter gzip'ed include objects which should not be, we gunzip them, but when we encounter gunzip'ed objects which should be, we gzip them, but only at compression level zero.

So in order to avoid unnecessary work, and in order to get maximum compression efficiency, you should:

```
sub vcl_miss {
        if (object needs ESI processing) {
                unset bereq.http.accept-encoding;
        }
}

sub vcl_fetch {
        if (object needs ESI processing) {
                set beresp.do_esi = true;
                set beresp.do_gzip = true;
        }
}
```

So that the backend sends these objects uncompressed to varnish.

You should also attempt to make sure that all objects which are esi:included are gziped, either by making the backend do it or by making varnish do it.

## 5.4 VCL Expressions

I have been working on VCL expressions recently, and we are approaching the home stretch now.

The data types in VCL are “sort of weird” seen with normal programming language eyes, in that they are not “general purpose” types, but rather tailored types for the task at hand.

For instance, we have both a TIME and a DURATION type, a quite unusual constellation for a programming language.

But in HTTP context, it makes a lot of sense, you really have to keep track of what is a relative time (age) and what is absolute time (Expires).

Obviously, you can add a TIME and DURATION, the result is a TIME.

Equally obviously, you can not add TIME to TIME, but you can subtract TIME from TIME, resulting in a DURATION.

VCL do also have “naked” numbers, like INT and REAL, but what you can do with them is very limited. For instance you can multiply a duration by a REAL, but you can not multiply a TIME by anything.

Given that we have our own types, the next question is what precedence operators have.

The C programming language is famous for having a couple of gottchas in its precedence rules and given our limited and narrow type repetoire, blindly importing a set of precedence rules may confuse a lot more than it may help.

Here are the precedence rules I have settled on, from highest to lowest precedence:

**Atomic** ‘true’, ‘false’, constants

function calls

variables

‘(‘ expression ‘)’

**Multiply/Divide** INT * INT

INT / INT

DURATION * REAL

**Add/Subtract** STRING + STRING

INT +/- INT

TIME +/- DURATION

TIME - TIME

DURATION +/- DURATION

**Comparisons** ‘==’, ‘!=’, ‘<’, ‘>’, ‘~’ and ‘!~’

string existence check (-> BOOL)

**Boolean not** ‘!’

**Boolean and** '&&'

**Boolean or** '||'

Input and feedback most welcome!

Until next time,

Poul-Henning, 2010-09-21

## 5.5 IPv6 Suckage

In my drawer full of cassette tapes, is a 6 tape collection published by Carl Malamuds "Internet Talk Radio", the first and by far the geekiest radio station on the internet.

The tapes are from 1994 and the topic is "IPng", the IPv4 replacement that eventually became IPv6. To say that I am a bit jaded about IPv6 by now, is accusing the pope of being religious.

IPv4 addresses in numeric form, are written as 192.168.0.1 and to not confuse IPv6 with IPv4, it was decided in RFC1884 that IPv6 would use colons and groups of 16 bits, and because 128 bits are a lot of bits, the secret '::' trick was introduced, to supress all the zero bits that we may not ever need anyway: 1080::8:800:200C:417A

Colon was chosen because it was already used in MAC/ethernet addresses and did no damage there and it is not a troublesome metacharacter in shells. No worries.

Most protocols have a Well Known Service number, TELNET is 23, SSH is 22 and HTTP is 80 so usually people will only have to care about the IP number.

Except when they don't, for instance when they run more than one webserver on the same machine.

No worries, says the power that controls what URLs look like, we will just stick the port number after the IP# with a colon:

> http://192.168.0.1:8080/...

That obviously does not work with IPv6, so RFC3986 comes around and says "darn, we didn't think of that" and puts the IPV6 address in [...] giving us:

> http://[1080::8:800:200C:417A]:8080/

Remember that "harmless in shells" detail ? Yeah, sorry about that.

Now, there are also a RFC sanctioned API for translating a socket address into an ascii string, getnameinfo(), and if you tell it that you want a numeric return, you get a numeric return, and you don't even need to know if it is a IPv4 or IPv6 address in the first place.

But it returns the IP# in one buffer and the port number in another, so if you want to format the sockaddr in the by RFC5952 recommended way (the same as RFC3986), you need to inspect the version field in the sockaddr to see if you should do

> "%s:%s", host, port

or

> "[%s]:%s", host, port

Careless standardization costs code, have I mentioned this before ?

Varnish reports socket addresses as two fields: IP space PORT, now you know why.

Until next time,

Poul-Henning, 2010-08-24

## 5.6 What do you mean by 'backend' ?

Given that we are approaching Varnish 3.0, you would think I had this question answered conclusively long time ago, but once you try to be efficient, things get hairy fast.

One of the features of Varnish we are very fundamental about, is the ability to have multiple VCL's loaded at the same time, and to switch between them instantly and seamlessly.

So Imagine you have 1000 backends in your VCL, not an unreasonable number, each configured with health-polling.

Now you fiddle your vcl_recv{} a bit and load the VCL again, but since you are not sure which is the best way to do it, you keep both VCL's loaded so you can switch forth and back seamlessly.

To switch seamlessly, the health status of each backend needs to be up to date the instant we switch to the other VCL.

This basically means that either all VCLs poll all their backends, or they must share, somehow.

We can dismiss the all VCL's poll all their backends scenario, because it scales truly horribly, and would pummel backends with probes if people forget to vcl.discard their old dusty VCLs.

### 5.6.1 Share And Enjoy

In addition to health-status (including the saint-list), we also want to share cached open connections and stats counters.

It would be truly stupid to close 100 ready and usable connections to a backend, and open 100 other, just because we switch to a different VCL that has an identical backend definition.

But what is an identical backend definition in this context ?

It is important to remember that we are not talking physical backends: For instance, there is nothing preventing a VCL for having the same physical backend declared as 4 different VCL backends.

The most obvious thing to do, is to use the VCL name of the backend as identifier, but that is not enough. We can have two different VCL's where backend "b1" points at two different physical machines, for instance when we migrate or upgrade the backend.

**The identity of the state than can be shared is therefore the triplet:** {VCL-name, IPv4+port, IPv6+port}

### 5.6.2 No Information without Representation

Since the health-status will be for each of these triplets, we will need to find a way to represent them in CLI and statistics contexts.

As long as we just print them out, that is not a big deal, but what if you just want the health status for one of your 1000 backends, how do you tell which one ?

The syntax-nazi way of doing that, is forcing people to type it all every time:

```
backend.health b1(127.0.0.1:8080,[::1]:8080)
```

That will surely not be a hit with people who have just one backend.

I think, but until I implement I will not commit to, that the solution is a wildcard-ish scheme, where you can write things like:

```
b1                                # The one and only backend b1 or error

b1()                              # All backends named b1

b1(127.0.0.1)                     # All b1's on Ipv4 lookback
```

```
b1(:8080)                       # All b1's on port 8080, (IPv4 or IPv6)

b1(192.168.60.1,192.168.60.2)   # All b1's on one of those addresses.
```

(Input very much welcome)

The final question is if we use shortcut notation for output from varnishd, and the answer is no, because we do not want the stats-counters to change name because we load another VCL and suddenly need disabiguation.

### 5.6.3 Sharing Health Status

To avoid the over-polling, we define that maximum one VCL polls at backend at any time, and the active VCL gets preference. It is not important which particular VCL polls the backends not in the active VCL, as long as one of them do.

### 5.6.4 Implementation

The poll-policy can be implemented by updating a back-pointer to the poll-specification for all backends on vcl.use execution.

On vcl.discard, if this vcl was the active poller, it needs to walk the list of vcls and substitute another. If the list is empty the backend gets retired anyway.

We should either park a thread on each backend, or have a poller thread which throws jobs into the work-pool as the backends needs polled.

The patternmatching is confined to CLI and possibly libvarnishapi

I think this will work,

Until next time,

Poul-Henning, 2010-08-09

## 5.7 Picking platforms

Whenever you write Open Source Software, you have to make a choice of what platforms you are going to support.

Generally you want to make your program as portable as possible and cover as many platforms, distros and weird computers as possible.

But making your program run on everything is hard work very hard work.

For instance, did you know that:

> sizeof(void*) != sizeof(void * const)

is legal in a ISO-C compliant environment ?

Varnish runs on a Nokia N900 but I am not going to go out of my way to make sure that is always the case.

To make sense for Varnish, a platform has to be able to deliver, both in terms of performance, but also in terms of the APIs we use to get that performance.

In the FreeBSD project where I grew up, we ended up instituting platform-tiers, in an effort to document which platforms we cared about and which we did love quite as much.

If we did the same for Varnish, the result would look something like:

### 5.7.1 A - Platforms we care about

We care about these platforms because our users use them and because they deliver a lot of bang for the buck with Varnish.

These platforms are in our "tinderbox" tests, we use them ourselves and they pass all regression tests all the time. Platform specific bug reports gets acted on.

*FreeBSD*

*Linux*

Obviously you can forget about running Varnish on your WRT54G but if you have a real computer, you can expect Varnish to work "ok or better" on any distro that has a package available.

### 5.7.2 B - Platforms we try not to break

We try not to break these platforms, because they basically work, possibly with some footnotes or minor limitations, and they have an active userbase.

We may or may not test on these platforms on a regular basis, or we may rely on contributors to alert us to problems. Platform specific bug reports without patches will likely live a quiet life.

*Mac OS/X*

*Solaris*.

Yes, we'd like to bump Solaris to tier-A but I have to say that the uncertainty about the future for OpenSolaris, and lack of time to care and feed the somewhat altmodishe socket-API on Solaris, does keep the enthusiasm bounded.

NetBSD, AIX and HP-UX are conceivably candidates for this level, but so far I have not heard much, if any, user interest.

### 5.7.3 C - Platforms we tolerate

We tolerate any other platform, as long as the burden of doing so is proportional to the benefit to the Varnish community.

Do not file bug reports specific to these platforms without attaching a patch that solves the problem, we will just close it.

For now, anything else goes here, certainly the N900 and the WRT54G.

I'm afraid I have to put OpenBSD here for now, it is seriously behind on socket APIs and working around those issues is just not worth the effort.

If people send us a small non-intrusive patches that makes Varnish run on these platforms, we'll take it.

If they send us patches that reorganizes everything, hurts code readability, quality or just generally do not satisfy our taste, they get told that thanks, but no thanks.

### 5.7.4 Is that it ? Abandon all hope etc. ?

These tiers are not static, if for some reason Varnish suddenly becomes a mandatory accessory to some technically sensible platform, (zOS anyone ?) that platform will get upgraded. If the pessimists are right about Oracles intentions, Solaris may get demoted.

Until next time,

Poul-Henning, 2010-08-03

## 5.8 Security barriers in Varnish

Security is a very important design driver in Varnish, more likely than not, if you find yourself thinking “Why did he do _that_ ? the answer has to do with security.

The Varnish security model is based on some very crude but easy to understand barriers between the various components:

```
              .-->- provides ->---------------------------------------.
              |                                          |            |
     (ADMIN)--+-->- runs ----->---.                      |            |
              |                   |                      |            |
              |-->- cli_req -->---|                      v            v
              '--<- cli_resp -<---|                     VCL        MODULE
                                  |                      |            |
     (OPER)                       |                      |reads       |
       |                          |                      |            |
       |runs                      |                      |            |
       |      .-<- create -<-.    |    .->- fork ->-.    v            |
       v      |->- check -->-|-- MGR --|            |-- VCC <- loads -|
      VSM     |-<- write --<-'    |    '-<- wait -<-'    |            |
     TOOLS    |                   |                      |            |
       ^      |     .-------------'                      |            |
       |      |     |                                    |writes      |
       |reads |     |->- fork ----->-.                   |            |
       |      |     |->- cli_req -->-|                   |            |
      VSM ----'     |-<- cli_resp -<-|                   v            |
       |            '-<- wait -----<-|                VCL.SO          |
       |                             |                   |            |
       |                             |                   |            |
       |---->----- inherit --->------|--<-- loads -------'            |
       |---->-----  reads ---->------|                                |
       '----<----- writes ----<------|--<-- loads --------------------'
                                     |
                                     |
                                     |
         .--->-- http_req --->--.    |    .-->-- http_req --->--.
(ANON) --|                      |-- CLD --|                     |-- (BACKEND)
         '---<-- http_resp --<--'         '--<-- http_resp --<--'
```

(ASCII-ART rules!)

### 5.8.1 The really Important Barrier

The central actor in Varnish is the Manager process, “MGR”, which is the process the administrator “(ADMIN)” starts to get web-cache service.

Having been there myself, I do not subscribe to the “I feel cool and important when I get woken up at 3AM to restart a dead process” school of thought, in fact, I think that is a clear sign of mindless stupidity: If we cannot get a computer to restart a dead process, why do we even have them ?

The task of the Manager process is therefore not cache web content, but to make sure there always is a process which does that, the Child “CLD” process.

That is the major barrier in Varnish: All management happens in one process all actual movement of traffic happens in another, and the Manager process does not trust the Child process at all.

The Child process is in a the totally unprotected domain: Any computer on the InterNet "(ANON)" can connect to the Child process and ask for some web-object.

If John D. Criminal manages to exploit a security hole in Varnish, it is the Child process he subverts. If he carries out a DoS attack, it is the Child process he tries to fell.

Therefore the Manager starts the Child with as low priviledge as practically possible, and we close all filedescriptors it should not have access to and so on.

There are only three channels of communication back to the Manager process: An exit code, a CLI response or writing stuff into the shared memory file "VSM" used for statistics and logging, all of these are well defended by the Manager process.

### 5.8.2 The Admin/Oper Barrier

If you look at the top left corner of the diagram, you will see that Varnish operates with separate Administrator "(ADMIN)" and Operator "(OPER)" roles.

The Administrator does things, changes stuff etc. The Operator keeps an eye on things to make sure they are as they should be.

These days Operators are often scripts and data collection tools, and there is no reason to assume they are bugfree, so Varnish does not trust the Operator role, that is a pure one-way relationship.

(Trick: If the Child process us run under user "nobody", you can allow marginally trusted operations personel access to the "nobody" account (for instance using .ssh/authorized_keys2), and they will be able to kill the Child process, prompting the Manager process to restart it again with the same parameters and settings.)

The Administrator has the final say, and of course, the administrator can decide under which circumstances that authority will be shared.

Needless to say, if the system on which Varnish runs is not properly secured, the Administrator's monopoly of control will be compromised.

### 5.8.3 All the other barriers

There are more barriers, you can spot them by following the arrows in the diagram, but they are more sort of "technical" than "political" and generally try to guard against programming flaws as much as security compromise.

For instance the VCC compiler runs in a separate child process, to make sure that a memory leak or other flaw in the compiler does not accumulate trouble for the Manager process.

Hope this explanation helps understand why Varnish is not just a single process like all other server programs.

Poul-Henning, 2010-06-28

## 5.9 What were they thinking ?

The reason I try to write these notes is the chinese wall.

Ever since I first saw it on a school-book map, I have been wondering what the decision making process were like.

We would like to think that the emperor asked for ideas, and that advisors came up with analyses, budgets, cost/benefit calculations and project plans for various proposals, and that the emperor applied his wisdom to choose the better idea.

But it could also be, that The Assistant to The Deputy Viceminister of Northern Affairs, edged in sideways, at a carefully chosen time where the emperor looked relaxed and friendly, and sort of happend to mention that 50 villages had been sort of raided by the barbarians, hoping for the reply, which would not be a career opportunity for The Assistant to The Assistant to The Deputy Viceminister of Northern Affairs.

And likely as not, the emperor absentmindedly grunted "Why don't you just build a wall to keep them out or something ?" probably wondering about the competence of an administration, which could not figure out to build palisades around border villages without bothering him and causing a monument to the Peter Principle and Parkinssons Law to be built, which can be seen from orbit, and possibly from the moon, if you bring your binoculars.

If somebody had written some notes, we might have known.

Poul-Henning, 2010-05-28

## 5.10 Did you call them *autocrap* tools ?

Yes, in fact I did, because they are the worst possible non-solution to a self-inflicted problem.

Back in the 1980'ies, the numerous mini- and micro-computer companies all jumped on the UNIX band-wagon, because it gave them an operating system for their hardware, but they also tried to "distinguish" themselves from the competitors, by "adding value".

That "value" was incompatibility.

You never knew where they put stuff, what arguments the compiler needed to behave sensibly, or for that matter, if there were a compiler to begin with.

So some deranged imagination, came up with the idea of the `configure` script, which sniffed at your system and set up a `Makefile` that would work.

Writing configure scripts was hard work, for one thing you needed a ton of different systems to test them on, so copy&paste became the order of the day.

Then some even more deranged imagination, came up with the idea of writing a script for writing configure scripts, and in an amazing and daring attempt at the "all time most deranged" crown, used an obscure and insufferable macro-processor called `m4` for the implementation.

Now, as it transpires, writing the specification for the configure producing macros was tedious, so somebody wrote a tool to...

...do you detect the pattern here ?

Now, if the result of all this crap, was that I could write my source-code and tell a tool where the files were, and not only assume, but actually *trust* that things would just work out, then I could live with it.

But as it transpires, that is not the case. For one thing, all the autocrap tools add another layer of version-madness you need to get right before you can even think about compiling the source code.

Second, it doesn't actually work, you still have to do the hard work and figure out the right way to explain to the autocrap tools what you are trying to do and how to do it, only you have to do so in a language which is used to produce M4 macro invocations etc. etc.

In the meantime, the UNIX diversity has shrunk from 50+ significantly different dialects to just a handful: Linux, *BSD, Solaris and AIX and the autocrap tools have become part of the portability problem, rather than part of the solution.

Amongst the silly activites of the autocrap generated configure script in Varnish are:

- Looks for ANSI-C header files (show me a system later than 1995 without them ?)
- Existence and support for POSIX mandated symlinks, (which are not used by Varnish btw.)

- Tests, 19 different ways, that the compiler is not a relic from SYS III days. (Find me just one SYS III running computer with an ethernet interface ?)
- Checks if the ISO-C and POSIX mandated `cos()` function exists in `libm` (No, I have no idea either...)

&c. &c. &c.

Some day when I have the time, I will rip out all the autocrap stuff and replace it with a 5 line shellscript that calls `uname -s`.

Poul-Henning, 2010-04-20

## 5.11 Why Sphinx and reStructuredText ?

The first school of thought on documentation, is the one we subscribe to in Varnish right now: "Documentation schmocumentation..." It does not work for anybody.

The second school is the "Write a {La}TeX document" school, where the documentation is seen as a stand alone product, which is produced independently. This works great for PDF output, and sucks royally for HTML and TXT output.

The third school is the "Literate programming" school, which abandons readability of *both* the program source code *and* the documentation source, which seems to be one of the best access protections one can put on the source code of either.

The fourth school is the "DoxyGen" school, which lets a program collect a mindless list of hyperlinked variable, procedure, class and filenames, and call that "documentation".

And the fifth school is anything that uses a fileformat that cannot be put into a version control system, because it is binary and non-diff'able. It doesn't matter if it is OpenOffice, LyX or Word, a non-diffable doc source is a no go with programmers.

Quite frankly, none of these works very well in practice.

One of the very central issues, is that writing documentation must not become a big and clear context-switch from programming. That precludes special graphical editors, browser-based (wiki!) formats etc.

Yes, if you write documentation for half your workday, that works, but if you write code most of your workday, that does not work. Trust me on this, I have 25 years of experience avoiding using such tools.

I found one project which has thought radically about the problem, and their reasoning is interesting, and quite attractive to me:

1. TXT files are the lingua franca of computers, even if you are logged with TELNET using IP over Avian Carriers (Which is more widespread in Norway than you would think) you can read documentation in a .TXT format.
2. TXT is the most restrictive typographical format, so rather than trying to neuter a high-level format into .TXT, it is smarter to make the .TXT the source, and reinterpret it structurally into the more capable formats.

In other words: we are talking about the ReStructuredText of the Python project, as wrapped by the Sphinx project.

Unless there is something I have totally failed to spot, that is going to be the new documentation platform in Varnish.

Take a peek at the Python docs, and try pressing the "show source" link at the bottom of the left menu:

(link to random python doc page:)

> http://docs.python.org/py3k/reference/expressions.html

Dependency wise, that means you can edit docs with no special tools, you need python+docutils+sphinx to format HTML and a LaTex (pdflatex ?) to produce PDFs, something I only expect to happen on the project server on a regular basis.

I can live with that, I might even rewrite the VCC scripts from Tcl to Python in that case.

Poul-Henning, 2010-04-11

CHAPTER

SIX

# VARNISH GLOSSARY

**backend** The HTTP server varnishd is caching for. This can be any sort of device that handles HTTP requests, including, but not limited to: a webserver, a CMS, a load-balancer another varnishd, etc.

**backend response** The response specifically served from a backend to varnishd. The backend response may be manipulated in vcl_fetch.

**body** The bytes that make up the contents of the object, varnishd does not care if they are in HTML, XML, JPEG or even EBCDIC, to varnishd they are just bytes.

**client** The program which sends varnishd a HTTP request, typically a browser, but do not forget to think about spiders, robots script-kiddies and criminals.

**header** A HTTP protocol header, like “Accept-Encoding:”.

**hit** An object Varnish delivers from cache.

**master (process)** One of the two processes in the varnishd program. The master proces is a manager/nanny process which handles configuration, parameters, compilation of :term:VCL etc. but it does never get near the actual HTTP traffic.

**miss** An object Varnish fetches from the backend before it is served to the client. The object may or may not be put in the cache, that depends.

**object** The (possibly) cached version of a backend response. Varnishd receives a reponse from the backend and creates an object, from which it may deliver cached responses to clients. If the object is created as a result of a request which is passed, it will not be stored for caching.

**pass** An object Varnish does not try to cache, but simply fetches from the backend and hands to the client.

**pipe** Varnish just moves the bytes between client and backend, it does not try to understand what they mean.

**request** What the client sends to varnishd and varnishd sends to the backend.

**response** What the backend returns to varnishd and varnishd returns to the client. When the response is stored in varnishd’s cache, we call it an object.

**varnishd (NB: with ‘d’)** This is the actual Varnish cache program. There is only one program, but when you run it, you will get *two* processes: The “master” and the “worker” (or “child”).

**varnishhist** Eye-candy program showing responsetime histogram in 1980ies ASCII-art style.

**varnishlog** Program which presents varnish transaction log in native format.

**varnishncsa** Program which presents varnish transaction log in “NCSA” format.

**varnishstat** Program which presents varnish statistics counters.

**varnishtest** Program to test varnishd’s behaviour with, simulates backend and client according to test-scripts.

**varnishtop** Program which gives real-time “top-X” list view of transaction log.

**VCL** Varnish Configuration Language, a small specialized language for instructing Varnish how to behave.

**worker (process)** The worker process is started and configured by the master process. This is the process that does all the work you actually want varnish to do. If the worker dies, the master will try start it again, to keep your website alive.

CHAPTER

SEVEN

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

$Id$

# INDEX

B

backend, 119
backend response, 119
body, 119

C

client, 119

H

header, 119
hit, 119

M

master (process), 119
miss, 119

O

object, 119

P

pass, 119
pipe, 119

R

request, 119
response, 119

V

varnishd (NB: with ’d’), 119
varnishhist, 119
varnishlog, 119
varnishncsa, 119
varnishstat, 119
varnishtest, 119
varnishtop, 119
VCL, 120

W

worker (process), 120