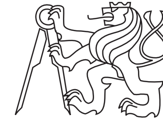
Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems



Bachelor's thesis

Improving Web Server Content Caching Performance

Tomas Kvasnicka

Supervisor: Ing. Jiri Smitka

11th May 2015

Acknowledgements

I would like to thank my whole family for overall life-time support, without them my studies would not be possible. I should also thank my friend Kristian Smith who helped correcting my English. Another important group of people are colleagues from work who added their comments to some of presented ideas - thank you guys. A great thanks also goes to Ing. Dana Vynikarová, Ph.D. for reading and correcting my whole thesis, for doing that in a very limited amount of time and for fast and comprehensive responding. In the end of course and obviously my supervisor. By this I sincerely thank you all.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on 11th May 2015 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Tomas Kvasnicka. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Kvasnicka, Tomas. *Improving Web Server Content Caching Performance.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Hlavním cílem této práce je výzkum a vylepšení webového serveru *nginx*. Tyto vylepšení přinášejí lepší výkon a vyšší použitelnost. Opravdovým přínosem této práce je vylepšená verze daného serveru nasazená v produkčním prostředí odbavující tisíce požadavků za sekundu.

Klíčová slova nginx, webový server, cacheování obsahu, výkonnost cache, HTTP protokol, C, Python

Abstract

The aim of this work is research of web server *nginx* and implementation of several improvements to this web server. These improvements present performance and usability increase. The real benefit of this work is patched version of *nginx* successfully deployed in production environment handling thousands of requests every second.

Keywords nginx, web server, content caching, cache performance, HTTP protocol, C, Python

Contents

Listings **xv**

Introduction **1**

1 About this thesis **3**
1.1 Exact problem formulation 3
1.2 Expected results 6
1.3 Thesis structure 6

2 Research of chosen web server **9**
2.1 Modular architecture and design patterns 9
2.2 Data structures and algorithms 16
2.3 Caching mechanism 23

3 Improvements implementation **29**
3.1 Ideas to discuss 29
3.2 Chosen improvement(s) 38
3.3 Realization 42

4 Testing, comparison, documentation **49**
4.1 Testing and production environment deployment 49
4.2 Comparison of original and patched version 52
4.3 Documentation and code maintenance 54

5 Outro **57**
5.1 Successes and failures until today 57
5.2 Plans for the near future 57

Conclusion **59**

Bibliography **61**

A Acronyms **63**

B Contents of enclosed CD **65**

List of Figures

1.1 *nginx* reverse proxy setup scenario 4

2.1 epoll-based event driven cycle 11

3.1 *nginx* cache file system layout 34

4.1 git repository layout 55

Listings

2.1 nginx.conf 13
2.2 ngx_http_foo_loc_conf_t 14
2.3 ngx_http_foo_srv_conf_t 14
2.4 ngx_http_foo_commands 15
2.5 ngx_http_module_t 15
2.6 ngx_module_t 16
3.1 ngx_http_proxy_help_store_info_from_header 46
3.2 ngx_http_upstream_proxy_t 47

Introduction

First of all, thank you all for being interested in this topic and for choosing my bachelor thesis as a source of information. I will try to provide you with the most relevant, accurate and up-to-date data.

This bachelor thesis focuses on improving content caching mechanisms in today's popular web server *nginx* (*name always written in lowercase*). The aim of the thesis is to show some important internals of *nginx*, design an improvement of the caching mechanism, implementing this improvement and properly testing it afterwards. This particularly means studying the data structures and algorithms *nginx* uses, then using this information for finding out how exactly the caching works and trying to improve it.

What we want to achieve here is at least one successfully implemented improvement to the caching mechanism. *Successfully* means it will be working, well documented & tested and *improvement to the caching mechanism* represents a piece of code that will improve the performance of the server. This improvement will make *nginx* deliver data faster in certain situations. It is also possible to develop additional usability/stability/scalability patches if valid reasons for such patches are found.

The major reason for creating these improvements is *nginx* research from the programmers point of view. We will also deploy our application on a worldwide content caching network and let it serve thousands of requests per second. Therefore every person in this world might benefit from using it by saving some precious seconds of their life every time they visit a website powered by this network.

Chapter 1

About this thesis

The first chapter of this thesis introduces the thesis itself. We are going to explain the nature of the problem, describe used configuration & possible improvements and also introduce the structure of the thesis. This is an extension to the introduction to make sure we all understand the situation and what will be happening in the following chapters.

1.1 Exact problem formulation

In this thesis we will focus on an important part of our overall Internet research - improving content caching mechanism of web server *nginx*. The reason for being interested in content caching is that a significant amount of data we see in our web browsers got there from some cache on the way to the origin web server and not from the server itself. Therefore it is only reasonable to make sure these caches respond quickly and with up-to-date data. *nginx* was chosen because it is second to none when it comes to performance and content caching. This is mostly because of the architecture and programming techniques it uses.

nginx is suitable for many different deployment scenarios including stand alone web server with support for dynamic content, mail proxy server and so on but what we are interested in is a reverse proxy configuration. When *nginx* is configured as a reverse proxy it stands in front of origin web servers and passes requests to them. During that it also caches the responses, so when another client requests the same URL *nginx* does not contact the origin web server anymore but answers directly from its cache.

1.1.1 Reverse proxy configuration

What this exactly means and how this really works is explained using a described example and a diagram below it.

Consider the following scenario: a client computer wants to download a HTML file `howto-be-rich.html` located at `www.awesomeweb.com/be/happy/` using the HTTP protocol. Its web browser therefore creates a HTTP request for `http://www.awesomeweb.com/be/happy/howto-be-rich.html` and then issues a DNS request for the domain `www.awesomeweb.com`. Now the DNS server containing records for this domain must be configured to respond with the IP address of the *nginx* server configured as a reverse proxy (NGX-RP) instead of the IP address of the origin web server serving files for `www.awesomeweb.com`. Based on the reply from the DNS server the client computer obviously sends its request to NGX-RP instead of sending it to the origin web server. NGX-RP listens for such requests and after receiving one of them it searches its cache for the response to that request. If such response is found and is still valid (note that the validity of records in cache is affected by various aspects such as time, type of response and so on) it is immediately returned to the client. If the cache does not contain the response or the response is no longer valid NGX-RP has to pass the request to the origin web server. To achieve this it knows this server's real IP address and so it simply re-issues the HTTP request from the client asking the origin web server for the response. This web server must always be capable of creating such response and therefore it sends it back to NGX-RP. NGX-RP now begins to save the response to the cache while simultaneously it forwards it back to the client. After this is done the response is cached & valid and the client has also received it. Now every other client asking for the same file will receive the answer much faster and without contacting the origin web server. [1]

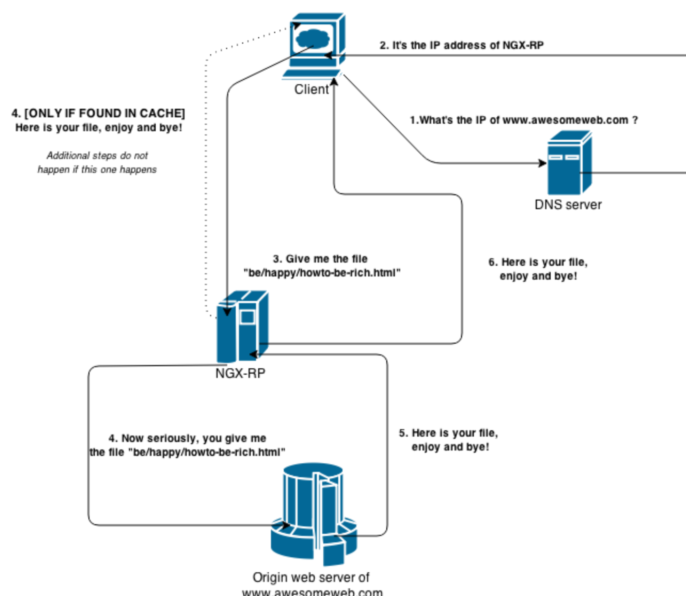


Figure 1.1: *nginx* reverse proxy setup scenario

1.1.2 Space for improvements

From the programmers point of view we can see that this is a very complicated & advanced process involving handling multiple network connections, disk operations, cache record's validity and a lot of other important things. Getting this all to effectively work together is definitely an interesting challenge. Although *nginx* is written with performance maximization in mind it does strictly stick to all related RFC's, tries to save a lot of system resources and was designed to work in all kinds of configurations, not just as reverse proxy cache. This offers us several improvement opportunities.

1.1.2.1 Network I/O

First thing that comes to our mind is reducing the number of network operations. We might achieve this by several techniques discussed later in this thesis but this is generally a very hard thing to do. Most of the methods described later are either already implemented or the opposite of that - too difficult to add. Generally we can say this involves modifying the served content on-the-fly which requires the modification to be very fast and very efficient to make it worth it.

1.1.2.2 Disk I/O

Disk operations are of course the slowest thing that happens in computer even when the machine is equipped with fast and powerful SSD drives. There is however at least some space for improvement although *nginx* already uses all advanced drive access methods like *aio(7)* or *sendfile(2)*. Ideas discussed later are dividing the cache between multiple drives without using LVM/RAID/ZFS, buffering temporary files only to memory and lowering the number of created temporary cache files. All of these ideas improve the performance of the server.

1.1.2.3 Cache manager

Cache manager algorithms are generally absolutely stable and well performing. Nevertheless, they might still offer a few places which can get upgraded. Instead of improving data structures or the algorithms themselves we could add some usability patches - in later parts of this thesis we will take a look at adding cache usage statistics and making the cache deanonymized. We will also talk about upgrades of the cache by presenting the stale-while-revalidate idea.

1.1.2.4 Others

This section includes ideas that do not fit anywhere else but are still considered interesting. There are a lot of ways how to improve *nginx* and most of them depend on used setup and needs. We will present some additional logging

options plus describe & modify HTTP range requests processing algorithm. *nginx* is not by default capable of logging communication with upstream origin web servers while it contains important information. Range requests are widely used by download managers or when watching videos and the way they are handled now is not ideal from reverse caching proxy setup point of view.

1.2 Expected results

The big picture end of this thesis is represented by patched, working, tested and deployed production ready version of *nginx*. This version is also better performing then the original implementation at least in one real life scenario and might present numerous usability and/or stability improvements.

When described in greater detail this obviously includes researching *nginx* source code at many places, getting familiar with its architecture and design, gathering all missing knowledge in the field of web server programming and several other prerequisities required to be able to start coding. After getting deeper into the application the process of the implementation itself starts. This usually comes in hand with a lot of unexpected and previously unseen problems which immediately require finding correct solutions in order to continue with solving the main problem. The part which follows consists of testing, debugging, fixing the mistakes and testing again. That is a structured, long-term process involving development of custom testing environment, debugging & tracing scripts and using numerous 3rd party tools. This part also includes the comparison of the patched version with the original one in a undoubtful way.

Starting with the analysis, the goal of that chapter is to understand the important parts of the code in *nginx* and being able to use this knowledge to develop our own upgrade. The implementation part will bring us one or more well designed patches which allow us to create a version that is better performing and preferably also more stable/usable/scalable. Last part of this thesis is expected to discover bugs created during the implementation part, resolve & fix them and provide a proof that the improvements are successful. All of this is expected to take several months of hard work to achieve.

1.3 Thesis structure

Here we will briefly describe the structure of this thesis. It is going to consist of three main parts: Analysis, Implementation and Testing. Each one of these parts is dedicated one full chapter with several subsections. Also each one of these parts could be covered in a separate thesis therefore we sometimes reduce the level of provided information only to the essentials.

1.3.1 Analysis

This will be the first part of this thesis coming right after this introduction. In this chapter we are going to cover analyzing *nginx* from the programmers point of view. Its content will provide us with information regarding detailed behavior of *nginx* in several situations. We have to focus on global modular architecture, the description of data structures & important algorithms and cache management. We might also take a look at some general information like source code structure, event-driven model and process separation. After finishing this part we are going to be able to understand major internal mechanisms, we will have a detailed idea of how *nginx* works and we will also be able to imagine what it might take to implement a working improvement.

1.3.2 Implementation

We will continue with part dedicated to the implementation. This is the core of this thesis as the most important parts regarding the implementation itself will be contained here. At the beginning of this part we are going to start with interesting upgrade ideas. Afterwards comes choosing one or more of these ideas along with suggestions of their design. And then we continue with the implementation itself along with the description of arising problems.

1.3.3 Testing

In the last part of this thesis we are going to focus on testing and comparison with the original implementation. Those are two necessary steps of every program development even though they do not actually bring any real usage to the user of the application. We will focus on creating an advanced script called *AMAST* for parallel requests launching. This script will in combination with memory usage checking tool *valgrind* give us relevant results about performance and stability of the patch. After making sure everything works exactly as expected we will manually compare the results obtained by *AMAST* and other testing tools to make conclusions about developed upgrade.

Chapter 2

Research of chosen web server

In the second chapter of this thesis we are going to focus at analysis of existing code. This is the research part of this thesis.

In the first section we are going to take a look at the overall design and structure of the sources. The description of *nginx* as a whole belongs here. Then we focus on used data structures and algorithms. In the end we will take a look at several important facts regarding the cache.

2.1 Modular architecture and design patterns

2.1.1 Design

First we are going to describe *nginx* from the programmers point of view to get some general understanding about what is it composed of. Then we focus on the most important design pattern it uses and try to explain its principles. This gets followed by the types of processes *nginx* forks and their functionality. In the end we also describe the structure of a simple configuration file.

2.1.1.1 General info

Information contained in this paragraph rephrases [2]. This program's numbers look very interesting even to an experienced programmer. *nginx* consists of approximately 2000 functions which manipulate the values of circa 140 custom structures and type definitions. All of this is happening in 280 source code files which together have just under 170 000 lines of C[3] code. It has been developed for more then 13 years and during that time it has been well tested on 12 operating systems using 9 different processor architectures. During March 2015 *nginx* was used in 21.22% cases someone visited one of the top million busiest sites. It is statically linked, uses numerous pre-processor conditions to avoid compilation of unnecessary code, inlines cpu specific assembly and

uses OS specific APIs. It also has almost no comments but keeps a very readable and clear coding style. On the other hand, the orientation in the code is very complicated due to the fact that everything is based on calling particular handlers at particular places of the code. This makes it almost impossible for IDEs to be able to navigate through the execution flow. *nginx* can also brag with requiring only about 250 bytes of memory per non-active keep-alive connection and 140 bytes per file in cache. Here we end our *nginx* summary and show some more of *nginx*'s design.

2.1.1.2 Event-driven model

As the parallelism in computer science is getting more and more natural we are looking for more ways to make programs do something in the background while doing something else in the foreground. This can be achieved by many ways and a lot of applications usually stick to forking multiple processes or at least creating multiple threads. While this can be an easy and quick way to achieve our goal, in extreme conditions it reveals its drawbacks. Context switching and unnecessary scheduling can slow down the whole application when being under heavy load even with today's hardware.

nginx handles this problem differently. Instead of creating enormous amount of processes or threads it uses event-driven programming techniques. This means *nginx* parametrizes most system calls to be non-blocking and thanks to that it almost never waits for any hardware operations that actually take some time. It also gets notified by the kernel when there is data ready to be processed, quickly processes it and returns to waiting for more data. This gets best described on a example. As a event driven mechanism we will use Linux *epoll(7)* but this can be easily replaced by *kqueue(2)*, *select(2)*, *eventports* or any other method[4] supported by *nginx*.

The master process described in section below first calls the typical combination of *socket(2)* - *bind(2)* - *listen(2)* in order to create a socket, bind to a specific address:port combination and make the socket listening for incoming connections. The only difference is that the socket is marked as *O_NONBLOCK*. This pattern usually continues with *fork(2)* or *pthread_create(3)* followed by *accept(2)* where the caller gets blocked until new connection gets accepted. Event-driven pattern however dictates something else. Forking stays the same, at least one worker process gets created but instead of getting blocked in *accept(2)* it calls *epoll_create(2)* followed by *epoll_ctl(2)*. This way it adds the listening socket to a list of sockets which should be watched. Then it calls *epoll_wait(2)* which blocks the caller until an event happens on one of the watched sockets. As soon as the first event on this socket happens it calls *accept(2)*, marks the newly returned socket as *O_NONBLOCK* and adds this socket to be watched by *epoll(7)* . Then it handles part of this commu-

nication but only until some *read(2)* or *write(2)* operation returns *EAGAIN* or *EWOULDBLOCK*. These return values signalize the kernel buffer for this socket is empty and the call would block. When this happens the worker returns to calling *epoll_wait(2)* until some data is ready for processing again.

This process then gets repeated over and over again until the program is terminated. It should be noted that *accept(2)* is still called only once for each connection - *nginx* knows the listening socket number therefore *accept(2)* gets called only when data on this socket is received. Other sockets are unaffected by this as *nginx* keeps communication context for every socket so that it knows which data it has already received/sent from/to this client. We should also mention that the word "socket" can be at most places interchanged for "file descriptor" as *epoll(7)* works well with any kind of underlying device. [5], [6]

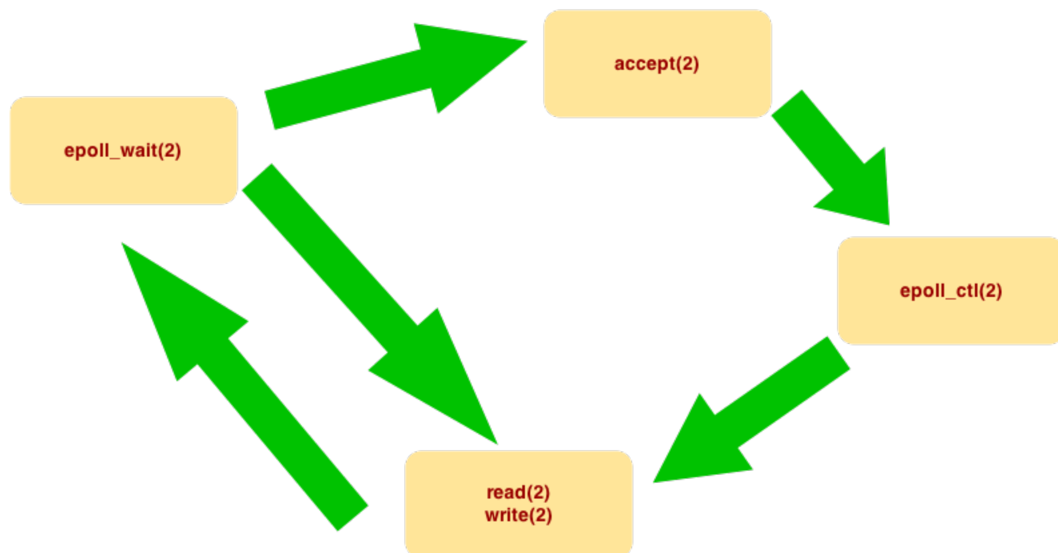


Figure 2.1: epoll-based event driven cycle

2.1.1.3 Processes

nginx consists of four main process types - master, worker, cache manager and cache loader. The instances of master and manager run for the whole life of the program and only as one process. Workers also run for the whole life but several instances get executed. Loader is a special case and it terminates itself after it is done with its job. Now we will describe them in greater detail.

- **Master** This is the process that gets executed first when you run *nginx*. It takes care about parsing the configuration and listening for incoming

connections. It also sets signal handlers and then forks the other processes and waits. This process must not die in any case as all its children depend on it. When a worker dies for whatever reason it gets restarted by master and only the connections handled by that worker are lost. The same goes for cache manager except for connection handling. But when a master gets killed or suffers from programming error *nginx* will not survive that.

- **Worker** Second most important process as without worker the HTTP server does not fulfill its function. Worker is the process responsible for accepting new connections and handling the whole communication with client. The number of workers is configurable and usually depends on the number of processor cores. The event driven loop inside each worker is extremely efficient thus one worker is capable of handling numerous connections without the need for context switching.

- **Cache manager** This is the process that only gets started in case we use *nginx* as a caching proxy. Generally we can say it takes care of managing the cache structure and keeping the cache in a state as requested by the configuration. *It gets described later in greater detail so take a look at section 2.3 for more information.*

- **Cache loader** Same as cache manager this process is also started only when we need content caching capabilities. Cache loader is different from other processes as it only runs for a limited amount of time. To stay brief we can say that it loads the content of the cache into the memory. *This process is also described later in a more detailed way so do not forget to take a look at section 2.3.*

2.1.1.4 Configuration file

To be able to understand algorithms parsing this file we need to see what it looks like. Configuration is divided into parts called blocks which are denoted by a keyword and curly brackets. Most important blocks are `main`, `http`, `events`, `server` and `location`. Block `main` is different as it does not have any keyword - anything put into the configuration file which is not inside any other block will be considered as being in `main` block. Each configuration directive has to specify in which blocks it is allowed to be and its handler. Handler then serves for parsing such a directive. Also, we are omitting configuration related to mail capabilities as that is not the subject of this thesis.

Block `server` can only appear in block `http` and block `location` can only appear in block `server`. Options configured in `http` and `main` blocks are then inherited by all `server` and `location` blocks unless overwritten in them. In `http` we specify general HTTP options, in server we define a virtual server and

location represents context specific to a particular part of request's URI. To make sure we all understand what this means we will show a piece of real configuration file. With this configuration, all virtual servers and all their locations will use *sendfile(2)* system call when copying data, but virtual server awesome.server.top will have this feature disabled when serving requests containing /videos/mp4/ in their URL.

```
worker_processes 1;           # this directive is in the~main block

events { use epoll; }         # events block

http                          # http block
{
  proxy_cache_path /cache/data levels=2:1 keys_zone=z1:2000m ...;
  sendfile on;

  server                      # server block
  {
    server_name  awesome.server.top;
    listen 443 default_server ssl spdy;
    listen 80 default_server;

    location /videos/mp4/
    {
      sendfile off;           # overwrites config from http block
      proxy_pass www.origin.com;
    }
  }
}
```

Listing 2.1: nginx.conf

2.1.2 Modules

nginx does not use objects as it is written in pure C[3]. It partially bypasses this disadvantage by using structures with function pointers and based on these structures it creates its own architecture of modules. *Information provided in following paragraphs was acquired by reading [7].*

Modules can do basically anything when it comes to handling a request but they are totally powerless when we need to change the core of *nginx*. To have a better understanding of what this means - teaching *nginx* how to seek in mp4 files can be done using a module, changing whether a file in cache is still valid or not has to be done in a different way. Modules are great when we need to add a new feature, but when we need to change current behavior of something essential they will not really help us. Now, we are going to take a look at their types and then their structures.

2.1.2.1 Types of modules

The main advantage of a module is that if you preserve its structure correctly and you do not call anything blocking you will not harm the event-driven pattern and thus everything will still be working as fast as possible. There exist 3 main types of modules - handlers, filters and load-balancers.

Handlers are the most important type of modules as generally they handle processing of a request. That means they for example handle connections to upstream servers, process CGI scripts or serve static files from drive. Typically one handler gets activated per request.

Filters on the other hand are a special case of modules - they get activated only when sending response to the client. Whenever *nginx* has data ready to be sent to a client it is passed to filters which have the possibility to alter it as required. They work in a chain thus all of them are activated for each response being sent, but typically a filter looks at the data it received and decides whether to process it or not. The order of their execution is defined at compile time with the filter which is actually sending the data to the client being always last. At the end of its execution the filter always calls the next filter in chain and returns its return value.

Load-Balancers are modules used only in certain situations. Basically only when *nginx* is talking to a upstream server (although not necessarily HTTP server) and it has more then one of those servers configured, those modules will get activated. Their main purpose is therefore to choose the appropriate upstream server. *nginx* implements two of those modules - round robin and hashing module. Round robin works as expected - distributes requests equally between upstream servers, hashing module assigns one particular upstream server to a set of IP addresses therefore the same client gets always connected to the same upstream server.

2.1.2.2 Module structure

Structure of a module is predefined by several structures. The first three hold the configuration of a module. These are called `ngx_http_<module name>_(main|srv|loc)_conf_t` based on whether they are for main, server or location blocks. The elements contained in those structures can be whatever the author of the module chooses and if the module does not require any configuration variables they do not need to exist at all. Here we show a simple location & server configuration of module foo.

```
typedef struct {
        ngx_str_t        location_prefix;
        ngx_uint_t       location_id;
} ngx_http_foo_loc_conf_t;
```

Listing 2.2: ngx_http_foo_loc_conf_t

```
typedef struct {
        ngx_str_t        server_prefix;
        ngx_uint_t       server_id;
} ngx_http_foo_srv_conf_t;
```

Listing 2.3: ngx_http_foo_srv_conf_t

Next we will show an example of structure `ngx_command_t` referring to previously showed configuration structures. *This structure is described later in greater detail in section 2.2.1.* We are going to take a look at the first element of the array. First is the name of the command, then its type, after that is the pointer to the handler of that command, then the two elements defining its memory position. Last is the pointer to custom data the command might need - NULL in our case. The handler is usually defined by the type of configuration variable - strings need to be set in a different way then integers. `ngx_null_command` is a special command which consists of all zeroes and by that *nginx* recognizes the end of this array.

```
static ngx_command_t ngx_http_foo_commands[] = {
    { ngx_string("location_prefix"),
      NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_str_slot,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_foo_loc_conf_t, location_prefix),
      NULL },

    { ngx_string("location_id"),
      NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_num_slot,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_foo_loc_conf_t, location_id),
      NULL },

    { ngx_string("server_prefix"),
      NGX_HTTP_SRV_CONF|NGX_CONF_TAKE1,
      ngx_conf_set_str_slot,
      NGX_HTTP_SRV_CONF_OFFSET,
      offsetof(ngx_http_foo_srv_conf_t, server_prefix),
      NULL },
      ...
      ngx_null_command
};
```

Listing 2.4: ngx_http_foo_commands

After we define commands and have structures to store their values, we need to connect our module with the rest of *nginx*. This has to be partially done by altering external configuration scripts and partially by specifying other important data types defining the structure of a module. Those two structures are called `ngx_http_module_t` and `ngx_module_t` and are also described in section 2.2.1. Therefore, the same as with commands, we will go directly to examples. These four functions will be called at appropriate places to create and merge configurations. Functions which create configurations usually allocate memory for configuration structures (`ngx_http_foo_loc_conf_t` & `ngx_http_foo_srv_conf_t`) and set default values in those structures. Functions for merging then mainly merge default values with those from a particular block.

```
static ngx_http_module_t  ngx_http_foo_module_ctx = {
    NULL,                            /* preconfiguration */
    NULL,                            /* postconfiguration */

    NULL,                            /* create main configuration */
    NULL,                            /* init main configuration */

    ngx_http_foo_create_srv_conf,    /* create server configuration */
    ngx_http_foo_merge_srv_conf,     /* merge server configuration */
```

```
    ngx_http_foo_create_loc_conf,   /* create location configuration */
    ngx_http_foo_merge_loc_conf     /* merge location configuration */
};
```

Listing 2.5: ngx_http_module_t

Last described structure will be `ngx_module_t`. This structure only holds all the previously defined functions & structures and makes the module cooperate with the rest of *nginx*. It also defines the type and version of the module and then leaves some space for padding - in the current implementation padding consists of zeroes. Version is used to reflect changes in this structure.

```
ngx_module_t  ngx_http_foo_module = {
    NGX_MODULE_V1,
    &ngx_http_foo_module_ctx,      /* module context */
    ngx_http_foo_commands,         /* module directives */
    NGX_HTTP_MODULE,               /* module type */
    NULL,                          /* init master */
    NULL,                          /* init module */
    NULL,                          /* init process */
    NULL,                          /* init thread */
    NULL,                          /* exit thread */
    NULL,                          /* exit process */
    NULL,                          /* exit master */
    NGX_MODULE_V1_PADDING
};
```

Listing 2.6: ngx_module_t

2.2 Data structures and algorithms

2.2.1 Data structures

Next thing to look at is important data structures. As we have seen there exist more then a hundred of them and therefore we cannot focus on every single one. We will reduce the amount of gathered knowledge and describe only the most significant ones. All the sizes are reported from x64 architecture. *All following knowledge was gained by studying [8].*

2.2.1.1 Request related

The first group of data structures is the one related to every request. This introduces a set of structures which gets usually allocated with an incoming request and also gets freed when the request is finalized. Now we are going to take a look at some of them.

- **ngx_connection_t** (*ngx_connection.h*) Main structures providing information about particular connection. These structures are 216 bytes big and they are reusable which means they do not get physically allocated and freed with every connection. They get created per network event. They have pointers to read/write events associated with this

connection, socket number connected with this connection, pool for allocated memory, log file, the IP address of the client, TCP parameters of the connection and many other items.

- **ngx_event_t** (*ngx_event.h*) One of the most low-level structures used in *nginx*. One of these exists per event known to the program. It has main flags symbolizing the properties of the associated event, void pointer to anything this event relates to and its handler. This structure has 104 bytes.

- **ngx_http_request_t** (*ngx_http_request.h*) These structures are HTTP connection specific and get created based on according `ngx_connection_t` structure. The size of this structure is 1320 bytes. Almost every function processing a request has access to this structure. Its items include arrays of configurations, the underlying connection, cache, several event handlers, pool for allocated memory, HTTP parameters of the request and various flags defining the properties of the request.

- **ngx_http_upstream_t** (*ngx_http_upstream.h*) A structure associated with every upstream server connection has the size of 1024 bytes. *nginx* uses these structures to fade out the differences between multiple upstream server types. The most important parts of this structure are pointers to functions which get called at predefined times during the communication with an upstream server. Also a very important flag which disables or enables caching gets stored here.

2.2.1.2 Configuration related

Another important group of data structures is the one related to parsing and afterwards storing the configuration. Those structures are either statically allocated or get stored during the initialization of *nginx*. In both cases, they stay in the memory for the entire time *nginx* is running. The most important ones will be described below.

- **ngx_cycle_t** (*ngx_cycle.h*) This is probably the most important data structure of all of them as it holds the whole configuration. Everything that is in the configuration file is represented here. Furthermore all the events are stored here, all the connections, all the opened files, shared memory and others. Generally we can say that functions having access to this structure have access to everything *nginx* currently has in its memory. This structure has 480 bytes.

- **ngx_command_t** (*ngx_conf_file.h*) As mentioned above, configuration file contains directives - now also known as commands. Every directive which is valid in *nginx* configuration file has its command structure somewhere. This structure consists of only 6 elements - name of

the command (configuration files are basically just command names from *nginx's* point of view), type of the command (this specifies blocks in which the command is valid and how many arguments it has), handler for the command (function that gets called by parsing algorithm when this command is found in config file), offset of main/http/server/location configuration structure that contains this command's variable, offset of this variable from the beginning of this structure, void pointer to arbitrary data if the command needs them (usually NULL). Its size is 56 bytes.

- **ngx_http_module_t** (*ngx_http_config.h*) This structure defines the organization of a HTTP module in *nginx*. It has 8 elements and all of them are function pointers. It enables the module to create and merge its configuration structures and to do whatever the module needs in the pre-configuration & post-configuration phase. These phases happen in the master process when parsing configuration of the http block and usually serve for allocating module variables. It has 64 bytes.

- **ngx_module_t** (*ngx_conf_file.h*) Another module defining structure, but this time it is not HTTP related. It consists of more general elements as even non-HTTP modules have to define this structure. This structure has several function pointers so that the module can call functions at several places of *nginx's* execution - in particular init/exit of a worker process/thread, init/exit of the master process and initialization of this module. Besides these pointers it holds the array of commands for this module, its configuration structures and several other things. As a proof of well designed application it has several allocated spare pointers and integers to be used in the future. This structure requires 200 bytes of memory.

2.2.1.3 Helper structures

As every other program *nginx* is also based on common data structures and algorithms. Therefore it uses many well known structures such as arrays, hash maps, red-black trees, heaps and so on. A brief description of some of these structures follows.

- **ngx_pool_t** (*ngx_palloc.h*) Default memory management structure. Basically every major structure has its own pool which it uses for allocating memory. These structures only hold pointers to allocated/free memory and to other pools because they work as a chained list. They also contain a handler for cleaning of the pool if the creator of the pool wishes to do so. Their size is 80 bytes.

- **ngx_array_t** (*ngx_array.h*) Arrays are used to store values next to each other at one location in the memory. This structure provides this for

any data type we choose. It only holds the number of allocated/total elements and the size of each element. It also has a pointer to the memory pool where the values will be stored. Its size is 40 bytes.

- **ngx_list_t** (*ngx_list.h*) Lists same as arrays have their memory pool and a few values. *nginx* implements a single linked list where all values have the same size. This helper structure requires 56 bytes of memory.
- **ngx_str_t** (*ngx_str.h*) This is the most simple structure we will describe, but it is still very useful. *nginx* stores strings as not null-terminated char arrays and their size. This is done to save repeated *strlen(3)* calls. This structure is used almost everywhere and uses 16 bytes.

2.2.2 Algorithms

Because not only data structures create programs, we have to take a look at used algorithms as well. It is obvious that *nginx* implements several hundreds of them and therefore as with data structures we can only focus at some of these. We will describe two of the most important ones in greater detail and then we will also briefly look at some of the more common ones. *As a source of following information was used [8].*

2.2.2.1 Main functionality

Algorithms listed here are very specific to *nginx* only and create the core of its functionality. They consists of calling numerous functions and interacting with all kinds of data structures. Knowledge of these algorithms will give us the possibility to alter *nginx*'s behavior at many important places.

- **Configuration parsing** First we are going to take a look at configuration parsing. *Please take a look at section 2.1.1.4 to make sure you understand how the configuration file is structured.* This algorithm is implemented in the master process and takes place right after parsing command line options, accessing CPU info and after the initialization of log files, the SSL engine, the time engine and several other support mechanisms. The most important function is `ngx_init_cycle` as that calls all the other parsing functions. Inside of it is a recursive function `ngx_conf_parse` which repeats the following process: call `ngx_read_token`, call handler for this token using `ngx_conf_handler`, if the token is a new block marked by curly brackets, call `ngx_conf_parse` again on this block. This results in reading a line, parsing that line and calling a handler for the directive at this line (if it is not a directive, *nginx* throws an error and exits). Special handlers we mention are those for `http` and `server` blocks - they lead to allocating configuration structures for each module in *nginx* as well as several other actions.

After parsing it continues with merging main/http/server/location configurations. This means directives specified in main get overwritten by directives in http, those get overwritten in server and those get overwritten in location block. This way we are able to specify a default setting of a directive at the http level and then change it per server/location as we need while settings for other servers/locations remain intact. We also have to mention that if a directive is not in the configuration file at all, it has its default value as specified by *nginx* documentation.

- **Request processing** Firstly, this is the core of every HTTP server and it is also the most complicated algorithm in *nginx*. It can behave in many different ways based on the progress of network communication, disk speed, configuration and several other important factors. Generally we can say that after parsing the configuration, *nginx's* workers just keep repeating this algorithm until they are terminated. Because of the fact that this algorithm is so variable, we will only describe one of its forms - with a file already in cache and with the *sendfile(2)* system call turned on. Also, we will have only one worker process and we will receive all the request data in just one *recv(2)* call .

  In the initialization phase of a worker process all listening sockets get registered to be watched by *epoll(7)*. They also get marked as listening so that *accept(2)* gets called when they are ready to be processed. After the worker finishes its initialization it jumps into a never ending loop. Here it does several checks to make sure it is not in the middle of exiting and if it passes them it calls the most important function of all of *nginx* - `ngx_process_events_and_timers`. This is the only function that uses blocking call on purpose. That call is *epoll_wait(2)* and it returns numbers of file descriptors that can be processed without blocking. Each of these file descriptors is encapsulated into `ngx_connection_t` structure and has its handler. `ngx_process_events_and_timers` then does several other checks and if they are passed it calls these handlers. We will now describe the function of the two most important handlers - `ngx_event_accept` and `ngx_http_request_handler`.

  - **ngx_event_accept** This handler gets called in case of a listening socket. That means a new connection has been initiated so *accept(2)* has to be called. After that the newly returned file descriptor by *accept(2)* is added to the list of file descriptors being watched by *epoll(7)*. Then a handler from `ngx_listening_t` structure is called - it was set to `ngx_http_init_connection` during configuration parsing. Initializing the connection sets a handler for related read event to `ngx_http_wait_request_handler` and then it also calls it. `ngx_http_wait_request_handler` is the func-

tion which calls *recv(2)* and stores received data in a buffer. If all the data gets received it also calls `ngx_http_create_request` which creates the `ngx_http_request_t` structure itself. In the end it also changes the handler of related read event and immediately calls it. The event handler is changed to `ngx_http_process_request_line` which tries to parse request's URI and if that is successful it continues with parsing the headers - each header also has its handler to parse it. At its end it calls `ngx_http_process_request` which sets handlers in read/write events to `ngx_http_request_handler`, then it sets several other handlers and then calls `ngx_http_handler`. `ngx_http_handler` changes previously set event's read handler to `ngx_http_core_run_phases` and calls it. This calls several functions each one representing one phase of processing a request.

These phases are responsible for handling the authentication, setting values to variables and most importantly responding to the request with correct content. The phase responsible for this is represented by `ngx_http_core_content_phase` which firstly calls function from added 3rd party module - purge. Unless the HTTP method of this request is the same as configured in the configuration of purge module it calls `original_handler`. The value of this function has been set to `ngx_http_proxy_handler`. `ngx_http_proxy_handler` represents the beginning of the proxy module as it tries to allocate `ngx_http_upstream_t` structure and sets all important upstream communication related handlers to functions from this module. First handler is `ngx_http_upstream_init` which adds connection's write event to be watched by *epoll(7)* and calls a function for initialization of request to origin - `ngx_http_upstream_init_request`. This function uses `ngx_http_upstream_cache` to find out the status of the file in cache - in our case this status will be hit. Based on the hit status `ngx_http_upstream_cache_send` will be called. This function starts sending out the requested data. First it uses `ngx_http_send_header` to send the headers of the response and then it calls `ngx_http_output_filter` to start processing the body by output filters. The chain of output filters depends on installed & compiled modules. The last one of them is `ngx_http_write_filter` which executes the actual writing function - *sendfile(2)*, *send(2)*, *write(2)* or *writev(2)*. With that the handler exits and *nginx* returns to waiting for new ready events.

- **ngx_http_request_handler** This handler on the other hand is only called on already accepted connections that require more data to be transferred. This is simpler then accepting the connection as the data only has to be read, processed by output filters and

send to the client. Because of this fact, this handler directly calls `ngx_http_output_filter` and the rest is the same as in the previous handler - all the filters get called with the last one writing the data to the output file descriptor. Thanks to *sendfile(2)* we only have to supply four things to do the sending - input file descriptor, output file descriptor, offset at which to start reading from input file descriptor and the number of bytes we want to send.

2.2.2.2 Helper algorithms

Apart from the two most important algorithms a series of those helping them must also exist in *nginx*. Some of those which we consider interesting will be described, but a significant number of others stay undocumented as they are out of this thesis's scope.

- **String parsing library** *nginx* can brag with wrappers for almost all standard string functions from *libc(7)*. Because it stores strings in `ngx_str_t` this is a very reasonable solution. All wrapping functions do not expect NULL-terminated strings and on the other hand require their sizes. This library is stored in *ngx_string.h* & *ngx_string.c* and implemented functions include wrappers for: *sprintf(3)*, *strstr(3)* or *strcasecmp(3)* and many others. There are also functions specific to *nginx* which can be used for parsing DNS names or printing contents of memory as hexadecimal strings.

- **Memory management library** To make it simple for upper level functions to work with memory, *nginx* uses a completely custom pool-based memory management. Provided functions are of course wrappers for *malloc(3)*, *calloc(3)* and other system functions but these are never called directly. This pool based memory management allows *nginx* to be able to allocate memory at almost any place in its code without worrying about freeing any of it. It also provides a small speed-up as it saves several system calls. Generally we can say *nginx* allocates pools of memory and then distributes memory from these pools by itself. When the whole pool is no longer needed it gets freed. This library lies in *ngx_palloc.h* & *ngx_palloc.c* and implements functions like: `ngx_create_pool`, `ngx_palloc`, `ngx_pcalloc` and many others. We must not forget to note that *nginx* also has its own implementation of the Slab algorithm used by Linux kernel.

- **Common data structures manipulators** As there are no objects in C[3], data structures are separated from algorithms altering them. This however does not mean these algorithms do not exist. *nginx* implements functions for all its basic data structures and access to these structures is done only by using these functions. These functions are always in

a separate source file which is related to a header file containing definitions of the particular data type. Typically these algorithms represent a function for initializing the structure and adding/removing/fetching an element.

2.3 Caching mechanism

2.3.1 Data structures and algorithms

Here we are going to describe several data structures and algorithms closely related to the caching mechanism. In structures we focus on the most important ones which describe the cache and its records. We will also take a look at two important algorithms realized by two special processes managing the cache. *Knowledge presented in following sections was obtained from [8].*

2.3.1.1 Cache related structures

All data structures related to cache are stored in *ngx_http_cache.h* file. Apart from the data structures, non-parametric macros defining cache lookup status codes are stored here.

- **ngx_http_file_cache_node_t** This structure represents one cache record loaded into memory. These nodes are stored using a red black tree which provides effective access to them. Nodes are defining the status of the record - if a file related to this record exists in cache, if the record is still valid, key for this record and several other. Memory for these structures is allocated using the Slab algorithm. Their size is 136 bytes.

- **ngx_http_cache_t** This structure represents the cache in the context of one request. That means it stores cache related information according to what is being served by this request. Therefore information from headers which might influence caching like `Vary` or `Etag` are here, the md5 hash of this request related cache file is here and also many flags defining several properties can be found here. Size of this structure is 584 bytes.

- **ngx_http_file_cache_header_t** Headers are structures which are physically written to the disk at the beginning of every regular file stored in cache. They are loaded back when the file is being read into memory. Therefore they contain data which are persistent and must be recovered after restarting *nginx*. This specifically means version of the header, how long is the file valid, when was it last modified and several other elements. These structures have 144 bytes.

- **ngx_http_file_cache_t** This structure represents the cache itself. It gets allocated when parsing configuration and contains among other

things shared memory for cache nodes, path to the root of the cache, number of files in it and important settings for cache loader. More then one of these structures can exist therefore they are stored in an array of every module providing caching capabilities. Size of these structures is 96 bytes.

2.3.1.2 Cache related algorithms

Now we are going to describe two main algorithms executed by processes taking care of cache management - cache manager and cache loader. First we will examine the algorithm which makes sure the cache has limited size and then we are going to take a look at loading files from drive to memory.

- **Limit cache size** Limitation for cache size is done using the max_size parameter of the proxy_cache_path directive. This value gets stored in the max_size variable of `ngx_http_file_cache_t`. This value gets periodically checked by cache manager process which is represented by the `ngx_http_file_cache_manager` function. This function gets executed based on timers expiration and processed as regular event. It does necessary synchronization and checks if the current size of cache is lower then configured maximum. If that is not the case it calls `ngx_http_file_cache_forced_expire` which does some checks and tries to delete the file. `ngx_cache_manager_process_handler` then adds a new timer to be processed later when it is expired with handler of this event set to itself.
- **Loading files into cache** This algorithm is always run only once per one instance of *nginx*. At its beginning it sleeps for 60 seconds before it starts - therefore the first minute of its life *nginx* is blind and does not know what it has in cache. On the other hand at the end of this algorithm funtion *exit(3)* is called terminating the process. Now to the algorithm itself. It checks the cache against being already loaded into memory or being in the middle of it. If this check is passed it the algorithm atomically swaps two values to indicate the cache is in the process of being loaded into memory. No other cache loading process can therefore touch it now. Handlers for files and directories are then set and function `ngx_walk_tree` is called to recursively walk through the cache and load files into memory. After this si done the function `ngx_http_file_cache_loader` returns and the cache loader process terminates.

2.3.2 Processes interacting with cache

In this section we will focus on processes interacting with cache, their spawning and functionality. First we are going to take a look at two managing processes

which make sure the cache has valid data & is in the right state and then we will take a look at process which is actually using the cache to store and load files. Those are the only processes accessing cache. *Please refer to [8] for more information.*

2.3.2.1 Cache manager & loader

Both these processes are started by calling the spawning function which is called inside of `ngx_start_cache_manager_processes`. This spawning function (`ngx_spawn_process`) uses system call *fork(2)* to spawn new processes and ten executes a function inside them. The function executed in both of these processes is the same, but the parameters of this function differs. That is also the only difference between loader and manager because one of these parameters is another function to be executed later and this function is different for these processes.

`ngx_cache_manager_process_cycle` is the first function called in the newly created processes. This function takes one argument - a structure, which is different for manager and loader. Both these processes then create a timer event but one of them sets its handler to `ngx_cache_manager_process_handler` while the other sets it to `ngx_cache_loader_process_handler`. Then both of them call the function `ngx_process_events_and_timers` which processes previously created timer events. This leads to calling their handlers which point to those two previously mentioned handler functions. This is the part where the function of loader and manager starts to differ.

Manager implements the previously mentioned limit cache size algorithm - it checks the current size of cache against its configured maximum and if its too big it deletes least recently used files. Loader on the other hand implements the loading files into cache algorithm and recursively walks through the cache from its root to its files creating cache nodes from the information found in cache headers and then exits.

2.3.2.2 Worker

Workers touch cache basically in two general cases - when they want to send response to client and when they want to save content received from upstream origin server. Both of these situations are happening when processing events in the main function `ngx_process_events_and_timers`.

First we can take a look at the situation when worker saves content received from upstream server. This happens when the file is not already in cache and caching is enabled by configuration. In that case *nginx* initializes the communication with upstream by calling `ngx_http_upstream_init_request` which

tries to search for the file in cache using `ngx_http_upstream_cache`. This function uses `ngx_http_file_cache_new` to create new `ngx_http_cache_t` structure specific for this request. `ngx_http_file_cache_open` then return miss status when looking for file related to this request. This sets various handlers and initiates a series of actions. Most notably after initiating communication with upstream it makes *nginx* to call `ngx_write_chain_to_temp_file` which saves data received from upstream to a newly created temporary file. This function is then called every time new read event from origin server is ready and writes data to drive.

What happens when the request is being served from cache is already described in section 2.2.2.1 in great detail. Here we therefore only summarize this algorithm. After calling `ngx_http_file_cache_open` a hit status is returned. This causes a totally different behavior as no communication with upstream server is initiated therefore no temporary file is created and on data received. Instead of it `ngx_http_upstream_cache_send` gets called and the response gets send to client. *Again, for detailed information regarding this please refer to section 2.2.2.1.*

2.3.3 File system layout

Here we will describe the hierarchy of directories and files in *nginx* cache. These descriptions fit for the actual version of *nginx* as this might get easily changed in the future.

As it is very hard to simultaneously read/write from/to file when the file is opened by multiple processes *nginx* presents the concept of multiple file types in cache. This is to avoid hidden synchronization issues. Also, to be able to quickly search for a file in cache *nginx* uses various techniques. One of them is renaming the file as its md5 hash and then using this hash when accessing various data structures. As a matter of fact, *nginx* creates file system structure based on these hashes. Now we are going to take a look at those two themes.

2.3.3.1 File types

First we are going to examine different file types existing in *nginx* cache. There are two main types of files in cache and one sub-type.

- **Regular file** These files represent standard files in cache. They get created from temporary files and their name is the md5 hash calculated from a user configurable string. At the beginning of these files a special *nginx* header exists which contains important internal data like file validity. After that the whole response from upstream server even with its headers is stored.

- **Temporary file** Temporary files get created every time a download from origin is started. They represent files which do not get read and exist only during the download. As soon as the download is finished they are moved/renamed to a regular file. Naming convention for temporary files is simple - their file names are increasing numbers. In case a file with such name already exists when trying to open it *nginx* simply generates new number which it thinks is free. Their purpose is to prevent processes from reading a file which is not completely downloaded yet. For performance reasons they must be at the same partition as regular files otherwise instead of just renaming/moving the file its content would have to be copied.

- **Non-persistent temporary file** These files on the other hand serve just as buffers. They balance the differences of client downloading speed and origin uploading speed, *but this gets more described in section 3.1.2.* They use the same naming scheme as normal temporary files but right after calling *open(2)* with flags signalizing to create the file a call to *unlink(2)* is made effectively deleting the file from the file system. However, as long as the file stays open data gets written to the disk and it serves as a normal file for the opening process. The difference is that after calling *unlink(2)* the file is inaccessible for the rest of the system - no standard system utilities are able to see the file (only *lsof*) and no other process is able to call *open(2)* on it therefore manipulate it in any way. This is also the way most programs create secured temporary files. After the connection for which this file was created is over the file gets closed and the system deletes it automatically.

2.3.3.2 Directory hierarchy

Directories created in cache have partially configurable structure. They are created inside cache root specified in the proxy_cache_path directive. Their purpose is to avoid a situation with too much files in one directory. Their name is always made of last 2 - 4 characters of the file name which is going to be stored inside them.

Therefore considering that a file name for a regular file in cache is always a md5 hash, last 2 - 4 characters of this hash are going to be candidates for names of directories leading to this file. Now how is this going to be done exactly defines the parameter levels of configuration directive proxy_cache_path. This parameter can have values 1:1, 1:2, 2:1 and 2:2. The first number represents the number of characters used for a name of a first level directory and the second does the same for the second level directory. This gets best described by a example so we will show all 4 variants with one file. [9]

- **levels=1:1**

  - filename: 068e03ec35d91fa1a1689a92a7b8
  - directories: $CACHE_ROOT/8/b/$FILENAME

- **levels=1:2**
  - filename: 068e03ec35d91fa1a1689a92a7b8
  - directories: $CACHE_ROOT/8/7b/$FILENAME

- **levels=2:1**
  - filename: 068e03ec35d91fa1a1689a92a7b8
  - directories: $CACHE_ROOT/b8/7/$FILENAME

- **levels=2:2**
  - filename: 068e03ec35d91fa1a1689a92a7b8
  - directories: $CACHE_ROOT/b8/a7/$FILENAME

Chapter 3

Improvements implementation

In this chapter we are going to focus on the actual improvements. We will follow the structure of chapter 1 and elaborate on ideas presented in that chapter.

Some of the ideas presented in chapter 1 will then get chosen and implemented, some will be marked as not so important/relevant and some of them will be discarded as not necessary. After the description of these ideas we are going to choose some of them for implementation. Their design along with major reasons for choosing them follows. In the end of this chapter we take a look at implementation details and arising unexpected problems.

3.1 Ideas to discuss

3.1.1 Network I/O

As stated before, reducing the number of network operations is not a trivia problem. Right at the start of this section we have to admit that we will unfortunately not be able to implement most of the ideas presented here. Some of them for the reason that they got already implemented while creating this thesis and some of them for the reason that they are already implemented although not in a ideal way and we do not have time possibilities to rewrite them.

We will however present one very important thought which is not necessarily impossible to add. Apart from lowering the number of transmitted packets this technique also reduces the number of write operations to the disk which is its main purpose (as disk write operations do actually really damage SSD drives, while transmission of packets does not do any harm to network controllers) - *therefore we will take a closer look at this later in section 3.1.2 and here just briefly present the draft.* Now lets take a look at the list of these thoughts.

- **Conditional revalidation request** When our research in this field of Internet traffic started the situation around *nginx* was different from today. Current stable version was approximately 1.4.7 at that time, nowadays version 1.7.12 has been released. One of the things that has changed since then is the support for conditional revalidation of files in cache. What this exactly means will be explained below.

  From chapter 2 we know that when *nginx* searches for a record in cache it then evaluates its validity. If this record is not valid anymore it crafts an HTTP request, sends that to the origin web server and forwards the response to the client while storing it to the drive. The question that comes to our minds is: Do we always need to rewrite the expired file with the fresh one? What if the content of the expired file is exactly the same as the content of the fresh file and it has just expired because of our default setting in configuration or some other reason? In that case it is obviously absolutely unnecessary to download the file again but we need a decision mechanism that will tell us if these two versions of the file are the same. Luckily HTTP provides such mechanism using conditional requests. As the server knows when the file was last modified it may include this information in a HTTP header when sending a response to a client. The client (whether it is a cache, another web server or web browser) may then cache this content and when the file is requested again it will only ask the web server if it has changed since the last time using the headers previously sent by the server. This means much less traffic then transferring the whole file therefore besides it saves resources it is also much faster.

  Prior version 1.5.7 *nginx* always downloaded the whole file after it has expired. This seemed as a potential space for upgrade as conditional requests were already known. We started with creating a simple prototype which only passed the headers necessary to do conditional requests and renewed the validity of the record. This showed up as not being ideal as renewing the validity required an additional field in the structure `ngx_http_file_cache_header_t` and also the client was required to sent the headers necessary to do conditional request. After these findings it also showed up that since version 1.5.7 *nginx* can handle conditional revalidation on its own without those drawbacks. Therefore we have decided only to upgrade *nginx* and cancel the research & development in this area.

- **Minify JavaScript/HTML/CSS** While transmitting source code files it is very common that they contain unnecessary characters like spaces, comments and so on. Those characters might get removed to

lower the number of bytes transmitted over the network. This idea is very simple but also very effective in case of large source files.

After quick search on the Internet we were able to find a module called *pagespeed* which can do exactly this. Unfortunately not in our setup. This module optimizes only files served directly by *nginx* when used as a web server. Files from cache however do not get manipulated and the size of *nginx* binary gets increased approximately 10 times when compiled with this module. Using *pagespeed* in such circumstances obviously does not bring any real benefits. An option is to re-implement the module and make it work with cached files. However, after reading its documentation and going through the source code the decision was made to abandon this. It is much bigger problem then expected and solving it too quickly will most definitely lead to a lot of potential bugs. Bugs are decreasing the stability and that is absolutely undesirable so we will to postpone the development of this patch because of limited time resources & knowledge. We might revert to it in the future after our knowledge of *nginx* gets even better then today and we are not pushed by forthcoming deadlines.

- **Unique download from upstream** The way *nginx* handles downloads from upstream servers might seem like resource wasting. When a request for a file not in cache is made its download gets started. However if another request for the same file is made while the download is in progress, *nginx* starts another download of the same file from the same server. This is logical as it still does not have the whole file in cache therefore it cannot send the data to the client immediately. But what *nginx* might do is skipping the part where it saves the data to the disk if it detects a download which is already in progress.

  As previously stated in chapter 2 *nginx* keeps two kinds of cache files - regular and temporary. When a download from upstream server is started the incoming data are stored in a temporary file before being moved to the cache itself. However, when there is a concurrent request for the same file which is initiated while the download of the first file is still in progress *nginx* creates another temporary file and starts to store the same data for the second time although it will discard this file as soon as the first one gets downloaded and moved to the cache. This naturally implies two negative factors - multiplying the amount of transferred bytes over the network and increasing the number of bytes written to disk. Unfortunately rapidly decreasing the amount of transmitted bytes is still very difficult to implement correctly as this requires reading/writing from/to the same file at the same time plus moving

the file while it is still being read. Those operations are not very safe especially in event-driven environment. Therefore we will mainly focus on reducing the number of bytes written to disk which will also theoretically decrease the number of transmitted packets as a side effect. *As this is more related to the disk I/O we are going to focus on this in the next section.*

3.1.2 Disk I/O

We are going to present three ideas which reduce the number of disk operations. One of these ideas will get solved by fine-tuning the configuration and therefore does not require any implementation. The rest of them are going to be ideas which are possible candidates for implementing although they will require some cooperation with other OS tools to be fully functional. Here is the list of them.

- **Disabling temporary files** As *nginx* is designed to be extremely resource saving it does not try to allocate a lot of memory in its default configuration. This behavior is required if our memory resources are limited but in case we have real-life production servers providing hundreds of gigabytes of memory available it seems only reasonable to override those settings.

  When *nginx* is forwarding files to the client it often creates special temporary files invisible to the administrator using common OS utilities (for example *ls*). These files are only serving as memory buffers for balancing differences between the upload speed of the upstream server and the download speed of the client. This might introduce a bottleneck of the communication if all participating computers have very fast Internet connection. Computers available for testing our improvements are very powerful servers which means we can try to disable this feature and use only RAM. After careful studying of *nginx* configuration we are going to find out that the directive proxy_max_temp_file_size is capable of doing exactly what we need when set to 0. As a result temporary files which will be moved to cache after the download is finished are still created but these special temporary files serving only for speed balancing are always only in memory thus saving numerous disk operations. Upgrading the configuration was enough in this case to get this idea to work.

- **Splitting cache between multiple drives** Sooner or later every administrator of a system running *nginx* as a proxy cache will be facing this problem: How do we split the cache among all available drives we have? The reasons for this are:

  – Running out of space and the need to make the cache bigger

- Disk being slow and network traffic being too high - the disk can not handle so much I/O operations at a time
- Not loosing all the data in case of disk failure
- Other

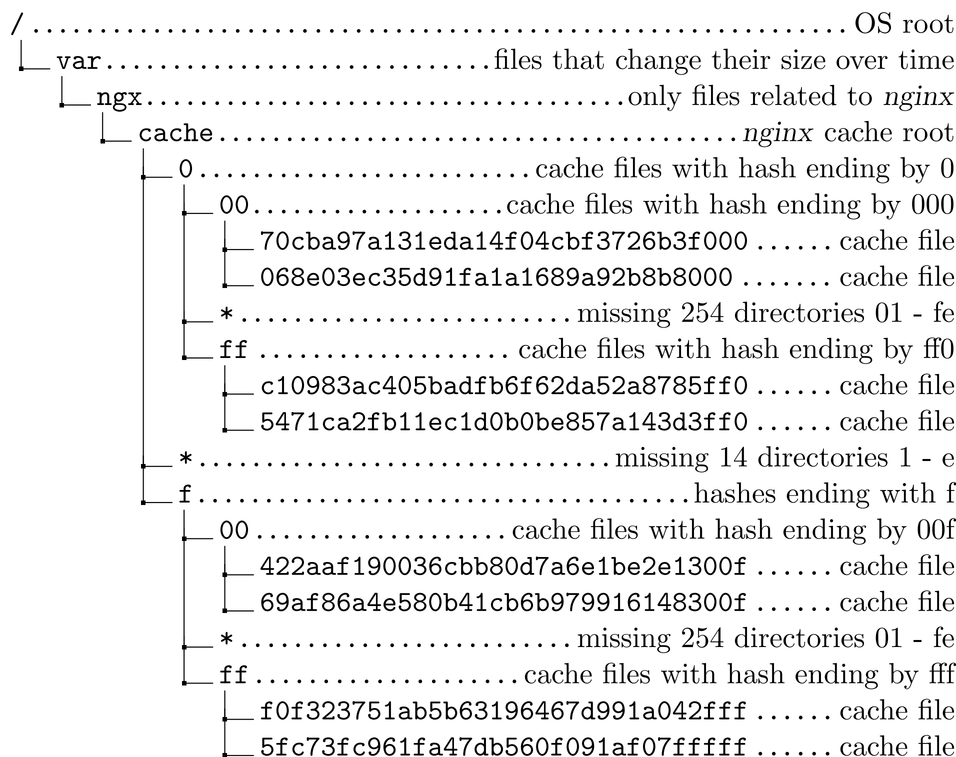
Apparently the first and naive solution is to add a layer of abstraction using LVM, add the drives to a volume group and then create one giant volume from this whole volume group and mount it as a cache. The obvious problem coming with LVM is disk failure - thing that very usually happens in production environment. In case we will use only LVM for this, the failure of one drive will necessarily mean loss of all the data in cache. This is certainly something we have to avoid. This can be done by adding some kind of RAID that will prevent these situations from happening. However, this is also not a preferred solution. Major argument against RAID is that cache does not have to be 100% replicated as the data still exist somewhere even in case of drive failure while the expenses of doubling every drive are significant (especially when using SSD). Also investing a lot of money into hardware RAID controllers so powerful to perform well in extreme conditions is also not desired and therefore we will focus on a different idea.

Although all known solutions for disk management have fallen us so far, we are still able to come up with a workaround which will not need anything more complicated from the OS then several mount-points. The way *nginx* organizes its cache *(take a look at chapter 2 section 2.3.3.2)* when configured with `proxy_cache_path /var/ngx/cache levels=1:2` creates a file-system layout which is partially shown on figure 3.1. Several missing directories are marked by * and cache files are used just as examples.

We can easily see that we have 16 directories (0 - f) and that we have 256 directories (00 - ff) in each one of them. Therefore we have 16 * 256 = 4096 directories which will contain files stored in cache. As the hash function distributes files into these directories relatively equally we are able to divide the cache between max 4096 drives also equally. We will probably have less then 4096 drives in which case we will simply use symbolic links to make more directories refer to one which will be our mount point.

The problem that comes with this is moving temporary files after the download is completed. In normal setup temporary files are on the same file-system as the cache so after downloading the file it is simply moved to the cache - moving is an inexpensive operation for OS. But if the file-systems (or drives) differ then the file gets copied which introduces extra

Figure 3.1: *nginx* cache file system layout



drive operations. Solving this problem will be the core of this idea if chosen for implementation.

- **Unique download from upstream** We already started presenting this idea in previous section. As write operations physically harm SSD drives we want to keep them at minimum - SSD drives, especially large ones, are still very expensive. Also, writing to drives is very slow therefore handling of such a request requires more time than handling request which does not need to write to disk. Therefore by eliminating disk writes during one request we effectively speed up all other requests happening at the same time. Downloading a file which content gets always discarded is therefore obviously unwanted. This is even more important in a situation of content caching network as cache misses are happening all the time as new origin servers cannot have their files in cache. Therefore with every new origin there is going to be a situation when the DNS servers already point to content caching network servers and those servers do not have a single file in cache.

Unfortunately there is no 100% reliable way in *nginx* how to tell if a file is already being downloaded. Thanks to the event-driven architecture

and multiple worker processes it is possible that one worker will initiate the download and some other will receive request for the same file - therefore such sign representing download in progress must be located in a shared memory zone and access to it must be protected using some synchronization primitive. Synchronization is as usually painful task to do but if we choose to implement this there is no way to avoid it.

3.1.3 Cache manager

Cache management is a complicated part of code in *nginx*. It includes synchronization, access to disk, validity checks, LRU algorithm and a several other problems. It is however the core of *nginx* when used as a caching reverse proxy. Therefore every patch introduced by us must not break its stability nor usability. We will now take a look at three ideas which might lead to such a patch.

- **Cache statistics** First presented thought are cache statistics. As previously indicated we will be testing *nginx* in real production environment. Such an environment introduces conditions for which *nginx* might not be fully prepared. This includes the size of cache in tens of TB and the number of virtual servers in hundreds. Now imagine the following situation: You run a worldwide content caching network and use *nginx* as a caching reverse proxy. Each virtual server in *nginx* represents one of your customers. These customers pay you based on how much traffic their visitors generate and based on how much data they have at your drives. Therefore you want to know which file in cache belongs to which virtual server to be able to determine the amount of money each customer owns you.

  Cache statistics are a solution to this problem. It is an improvement providing an easy way to access important information (server name, number of files in cache & their total size) per virtual server. If we choose to implement this we will be dealing with altering the storing mechanism. But considering the previously stated situation this is a essential patch to run such business.

- **Deanonymize cache** This is basically an advanced version of the cache statistics idea. Information provided by that improvement is definitely necessary but very likely it is not everything we might want to know about our cache. A lot of other useful information can be logged somewhere for later manipulation. Such examples might include - virtual server to which the file belongs to, HTTP request string, validity of the file, when it got stored (we also have to consider the possibility of being notified when file is about to get deleted), mapping between virtual

servers and some other entity (it might be clients as previously suggested) and so on. Lets presume we have patched *nginx* to provide all this info. Now we have to decide how and where to store it. Also presume we have cache of size 80 TB, approximately 25 servers and traffic on our servers does not drop below 50 Gbit/s which means frequent changes in cached files. As it is going to be a lot of data and we want to search them very quickly, be able to sort them by several criteria and probably something else a relational database seems to be the only reasonable solution.

Interacting with a database brings a lot of joy when everything is working, well tuned a excellently performing. But it also brings a lot of pain if something decides to stop working and such moments happen. As we do not want to decrease the stability of *nginx* in any way therefore making it dependant on a database is not considered safe. There are situations when this will be unavoidable (file deduplication between servers in the same datacenter for example) but for now we will not make *nginx* rely on an external application. We are also going to be able to gather a lot of useful information by using the upstream log patch - this is presented in the next section.

- **Stale-While-Revalidate** This idea was presented in [10] by Mark Nottingham during December 2007 and published as Internet-Draft in May 2008. The point is to always answer the client from the cache, even if the record is no longer valid while asynchronously updating it. This obviously brings a speed-up at the price of possibly seeing stalled content for a very low number of clients. After carefully studying *nginx* internals there are three ideas how to implement this improvement.

  - **Subrequest** Always mark the validity lookup as hit unless a request carrying special flag is detected, then connect to the origin and update the file. The special request is manually enforced using sub-request mechanism in *nginx* after all the data are send to the client. This works and is considerably easy to implement but has one important drawback - the connection with the client is not closed until the file is updated.
  - **External application** Also always mark the validity lookup as hit but if we know it was supposed to be stale, tell this to a external application which will reissue this request from local loopback interface. Obviously, requests coming from this interface never consider as hit. This is also quite easy to implement as it requires minimum changes in *nginx* itself and developing a simple TCP client/daemon application but it also has major drawbacks. Undesired overhead,

inter-process communication and making *nginx* dependant on external application.

– **Caching-only request** Create a special type of request that is handled in a different way then usually - it does not require a connected client, but only contacts the upstream server and saves the data to the cache. This request is to be issued every time a stalled answer is marked as hit and returned. Sounds like the optimal way to do it but thanks to *nginx* architecture (it creates request structures based on connection structures which are created based on *epoll(7)* event which is hardly to be faked) this is close to impossible to implement without much deeper knowledge of *nginx* then we currently have. Implementing this now represents a significant space for programming mistakes and that is absolutely undesirable.

As we can see, all those ideas have their disadvantages. Also there is a possibility in *nginx* to use stale record in case it is already updating. This with the combination of conditional revalidation led us to the decision that this idea will not get implemented.

3.1.4 Other

Last section of ideas will contain those that do not fit anywhere else. However they still bring important features.

- **Range request processing** Range requests are widely used by popular browsers even for fetching regular HTML files. *nginx* normally removes the HTTP range header and passes the request as a regular one. This causes the origin upstream server to return whole file instead of the required part which has two effects: *nginx* is able to cache the response and the client might be waiting for the response very long time in some cases. In a situation where the client requested only a small part of a file from its end and the file is very large it will have to wait until the whole file has been transmitted between the origin and *nginx*. In such cases having a proxy can be even slowing the connection down because sending the request directly to the origin would be handled properly as a range request.

  The idea is to alter this behavior in such a way that range requests will still be handled as range requests and if possible they also get cached. That means only the requested part is going to be transferred between the origin and the proxy but we will try to store the whole file to the drive. This provides a performance boost in several situations.

- **Upstream log** This idea is fairly straightforward but presents a very powerful feature. *nginx* is by default unable to log any communication with origin upstream servers. However such a communication contains a lot of important information we want to know (values of particular headers sent to origin, time it took to process the communication with origin, ...). We might therefore try to add a possibility to log this communication and then process these logs using external tools to gather required info.

3.2 Chosen improvement(s)

3.2.1 Range request processing

First chosen idea is to change the algorithm processing range requests. It is clearly a problem which needs to be solved in every setup using *nginx* as reverse cache. This is the first improvement of the performance of *nginx*.

3.2.1.1 Reason for improvement

As previously stated, the way *nginx* processes range request is not ideal in our setup. Reverse proxy cache always speeds up the communication. Maybe in several cases it might be transparent and do not adjust the speed at all but no way it can slow it down. Unfortunately, in cases where a client sends for example a header like `Range: bytes=-1024` by which is it asking for the last 1024 bytes of a file the client will encounter significant waiting. As range requests are also used by web browsers when watching videos this happens every time the client seeks to the end of the video. This leads to a scenario we want to eliminate and therefore we will try to design and implement a patch solving this problem.

3.2.1.2 Design

We want to keep the design of this simple and easy to make sure nothing breaks up. First, we want to make sure range requests are handled as range requests and therefore the origin knows about them and treats them that way. This ensures the client will not be waiting longer then it really has to. Second, if possible, we want to cache the data as if it was a normal request. That way we can still benefit from range request which caused a cache miss.

One way to implement this is to do a previously mentioned so called caching-only type of request. We will then issue one of those requests for each range request that is not in cache and handle the range request in a proper way. That gives us a way of quickly transmitting only required bytes and asynchronously downloading the whole file and storing it to the cache. It is obviously a source

of downloading some data twice but that is acceptable as the client is not waiting for them and it happens in the background. But as also previously mentioned implementing such a type of request is very complicated due to the architecture of *nginx* so we will try to find another solution.

We lower the condition for caching the data and admit that ranges into the middle of the file are impossible to cause proper caching without some drawbacks. Therefore we focus on ranges from the beginning of the file. We can easily recognize them as the header will have either this syntax `Range: bytes=0-*` (where * represents any valid integer number) or this syntax `Range: bytes=0` (where the client asks only for the first byte). Those ranges are going to cause downloading of the file from its beginning and simultaneously responding to the client with the downloaded content as it has also requested it from its beginning. So these requests cause proper caching and proper handling of range requests at the same time. An obvious drawback of this idea is that other requests then those starting with 0 will not lead to caching. This is however acceptable as popular browsers send such requests at the beginning of the communication using range requests - if we get back to previously mentioned video watching, Google Chrome sends request with range starting from zero every time we open a video even though we instantly seek to its end. Therefore this disadvantage can be ignored as it only has impact in a very limited amount of scenarios.

3.2.2 Unique download from upstream

Second thing to implement will be a patch ensuring we are downloading only one copy of a file from the upstream server at a time. This is also a performance upgrade.

3.2.2.1 Reason for improvement

Besides the obvious drawback of slowing down the communication between origin and *nginx* thus slowing down the communication between *nginx* and client the most important issue are write operations to SSD drives. Extreme amounts of write operations happening constantly for a significant period of time might physically damage the drives. Although this happens only in special scenarios we have to try to avoid it. When we are a content delivery provider we are in a situation where we have nonstop drive throughput in hundreds of megabytes. Here we want to decrease writing as much as we can.

3.2.2.2 Design

Our goal is to achieve a mechanism indicating that a file is currently being downloaded. Such a mechanism will help us in preventing situations where multiple clients request the same file which is not yet in cache and each of

these requests initiates a fresh download. Fortunately indication of download in progress is a YES/NO question and thus can be saved in just one bit. Such a bit must be stored in a memory zone which is accessible by all *nginx* worker processes and therefore access to it must be protected by some synchronization primitive. When synchronization is used we might easily end up in a situation where all workers think that the download is in progress while the only worker which was actually downloading it died for whatever reason. Because of that we need a mechanism to indicate that the download is not receiving any more data. That leads us to adding a timestamp indicating the last time this request received some data. Those two values obviously and inevitably increase memory usage but we have to accept that.

As for the algorithm itself, we will try to be as simple and effective as possible. Every time a request wants to access a resource which is not yet in cache we make a check if a download of this resource is already in progress. In case it is we disable caching and only forward the data from the origin to the client. That means only the first request of this resource will cause caching and the others will simply get served by the origin. We also have to update the timestamp every time the downloading request receives some data to signalize the download is still in progress. Although this might probably be bypassed and left upon some *nginx* timeout we want to be precise.

3.2.3 Cache statistics

Another chosen upgrade will be to the cache manager. We add a feature providing us with information about files in cache per virtual server.

3.2.3.1 Reason for improvement

Previously mentioned scenario where a customer is represented by its virtual server in *nginx* configuration is not so far away from a real life situation. Considering we are providing caching capabilities worldwide to everybody we will very likely have multiple customers having files in one server. Their caches need to be separated and their origin servers will differ so this is only reasonable. Also the idea of payments based (besides traffic) on the amount of used disk space is very natural in real life conditions. The outcome is that having such capability is a must in production environment.

3.2.3.2 Design

Requirements on this patch were following:

- if possible display the information in a human readable way without additional processing

- if possible display enough information to be able to determine which customer owns which virtual server without additional processing

- in case something goes wrong and we are unable to display what we want to, do not make this situation affect anything else

Two main problems were how to show the data to the user and how to store the data in case of loosing RAM content (which will inevitably happen). Web servers usually only display generated web pages and loosing RAM content means we will have to store our data to the hard drive which must be done carefully to do not break the event-driven architecture.

Fortunately *nginx* contains built-in *stub status* module which is normally used for displaying information about processed requests in real time. The module loads values of several variables and generates a HTML page containing these values. This HTML page is then returned upon request of a predefined URL. We will therefore alter this module to load values of additional variables in which we will store information about cache. Also, we are going to provide *nginx* with a binary file containing descriptions about particular virtual servers and display those descriptions as well.

As for accessing the drive, we mainly want to be sure we will not break the saving and loading scheme *nginx* uses. We can therefore change the naming convention and alter the names of files in cache to contain an ID of the virtual server to which they belong to. The ID is going to be specified as a value of a variable in *nginx* configuration file so *nginx* will know it when saving the file. This variable obviously has to be set per virtual host and loaded back into memory by cache loader process.

3.2.4 Upstream log

Last but not least implemented improvement is logging communication with upstream origin servers. *nginx* is totally missing this functionality while it is very useful when debugging problems or gathering information.

3.2.4.1 Reason for improvement

When talking to origin server we will mainly want to know if the communication was successful, how much data was transferred & how long did it take, what was the host name of the virtual server & the origin server and several other things. This info is going to help us when searching for major error causes as it easily shows if we received corrupted/bad data already from upstream. We can also generate usage statistics of origin servers.

3.2.4.2 Design

Due to the fact that logging capabilities are already in *nginx* the algorithm for this patch was not difficult to design. It was more about choosing what do we need to log and when then how exactly to do it. Therefore here we will just quickly describe the logic behind upstream log.

We add a configuration directive with its handler. The configuration directive has one argument and that is the path to the log file. If this directive is not specified there is no default value and no logging happens. To keep things simple we then just write the required information into this file when the processing is at the proper place - in this case it is at the end of communication with the upstream server. Required information was carefully chosen to include these values:

- HTTP method used for communication with the upstream server
- `Host` header value received from the client
- Requested URI
- HTTP status code received from the upstream server
- IP address & port of the upstream server
- `Host` header value being sent to the upstream server
- `Content-Type` header value received from the upstream server
- Timestamps of start/end of the communication with the upstream server
- Name of file in cache if the response is stored there

3.3 Realization

3.3.1 Range request processing

First we take a brief look about what we had to do to alter the behavior of range requests processing algorithm.

3.3.1.1 Implementation details

To be able to implement the design described in section 3.2.1.1 we have to mark those special range requests which are suitable for caching. We can add one bit flag into the `ngx_http_request_t` structure (*see chapter 2, section 2.2.1.1*) to mark such requests. Then we will focus at function `ngx_http_proxy_create_request` which is responsible for allocating and creating `ngx_http_request_t` structure of request being sent to the origin web

server. At the beginning of this function we are going to decide if we set the flag to on or off. This is done by checking that this is a range request, then checking that this request is cacheable and then comparing the content of the `Range` header against a predefined string. If all those checks are positive the flag is set to on. A set of conditions based on this flag follows. It turns off buffering & caching and also passes the `Range` header to the origin web server follows. Due to the license terms we are unfortunately unable to show the actual code here as it is currently being used and thus is a part of a trade secret.

3.3.1.2 Problems and complications

Although first analysis of *nginx* code did not show anything else which might require additional editing careful testing uncovered one hidden mistake. *nginx* by default processes range request using its own module for requests of this type and this was causing problems. Our testing environment compares md5 hashes of received data against the md5 hashes of data that were actually sent by *nginx* and those two values did not match. That led us to the conclusion that something must be seriously broken and launched a series of debugging. After we were successful in finding this mistake we released a patch fixing it.

3.3.2 Unique download from upstream

Now we will take a look at the implementation of another patch - unique download from upstream.

3.3.2.1 Implementation details

The first thing that must be altered is the `ngx_http_file_cache_node_t` structure where we add a one bit flag indicating download in progress and we also add a timestamp indicating last time the request received some data. Next we have to develop a function for checking if the download is in progress and if it has not died (`ngx_http_file_cache_downloading_start_check`), then a function for timestamp updates (`ngx_http_file_cache_downloading_update`) and finally a function to reset the flag in case the download has already ended (`ngx_http_file_cache_downloading_ended`). The first function has a series of conditions checking the one bit flag and comparing the timestamp plus some predefined time offset against actual time. Based on those values it either sets the downloading flag to 1 and returns that the download was just started or does not change the flag and its return value indicates the download is already in progress. Second function only updates the timestamp if it can be updated or resets the flag to 0 if the download has timeouted. The same goes for the last function except it does not update the timestamp and just checks if the download is not in progress anymore and resets the downloading flag accordingly.

The updating function then gets called every time this request receives some data as it is called inside of `ngx_http_upstream_process_request` which is the core of request processing. The checks are in `ngx_http_file_cache_open` which gets called for every accepted request, therefore in case of receiving two requests with the same URL the second one will be correctly detected as already in progress and caching will not get initialized. The function for resetting the flag is then called when the request is over.

3.3.2.2 Problems and complications

This patch was luckily implemented without any unexpected situations. This was because it was well designed and carefully implemented.

3.3.3 Cache statistics

As previously stated next implemented improvement are cache statistics. Below are the details of this patch and problems arising during its development.

3.3.3.1 Implementation details

We are facing multiple problems we need to solve in a correct way. First we take a look at changes in the *stub status* module. We will add a pointer to a initialization function of this module which will load the previously mentioned binary file containing descriptions. This file is organized as a map and the position of the description determines the ID to which that description belongs to. This is done to avoid parsing. Next in this module we change the way `ngx_http_stub_status_handler` behaves so that it allocates more memory and displays more information. The information to display will be loaded from added variables stored in `ngx_http_file_cache_sh_t` structure and also from previously loaded file. Added variables are two arrays, one for file sizes and one for the number of files in cache. The virtual server ID set in configuration file serves as an index into these arrays.

After editing *stub status* module we have to alter the naming scheme. This is done by changing `ngx_http_file_cache_name` to add a hash sign (#) into the file name as a separator between md5 hash and virtual server ID. Minor changes in `ngx_http_file_cache_add_file` and `ngx_http_file_cache_read` are also required to make them compatible with this new naming convention. Basically we just parse the file name and save the part after the separator into a variable. Finally, when a file gets added, updated or deleted we use ID assigned to this particular virtual server and alter the values in those two previously mentioned arrays.

3.3.3.2 Problems and complications

The first problem we encountered was just a little misunderstanding. *ls* shows different file sizes then those calculated in *nginx*. This caused repeating some tests more times then necessary but after we realized this it was quickly fixed by using *df* instead of *ls*. The only real trouble was with purging the files from cache. Purging uses a special purge module and it had no idea about our cache statistics patch so it did not update the stats correctly. This was revealed by *AMAST* testing and fixed by altering the purge module appropriately.

3.3.4 Upstream log

Finally we focus on upstream log patch development and the details regarding it. Problems and complications are mentioned as well.

3.3.4.1 Implementation details

As previously mentioned there is nothing really difficult about the logging algorithm and mainly it was already invented by original *nginx* developers. So we will just the same what they did only at different place with different values.

We add a logging function `ngx_log_upstream` which does the actual writing into log file. It also checks for variadic macros and behaves based on their presence but its outcome is always the same. Next we add a function `ngx_http_upstream_log_entry` which will be preparing the string to be written to log file. This function will do several checks against values that do not always have to be available and also some time calculations. Then it creates the line to be written and passes it to `ngx_log_upstream`. This is the function that we call when finalizing the communication with upstream server.

We also create `ngx_http_proxy_help_store_info_from_header` function to save the value of the `Host` header being sent to upstream. This function will be called from `ngx_http_proxy_create_request` because that is the place where we have easy access to buffer containing the value of this header. The value of the header will be later read by `ngx_http_upstream_log_entry` when creating the log line.

To be able to log all of this to a file we need a handler of upstream log configuration directive. This handler is called `ngx_http_core_upstream_log` and it opens the file passed to the configuration directive. If this operation is unsuccessful *nginx* dies while parsing its configuration.

3.3.4.2 Problems and complications

This patch was created without any serious issues but it was not done in a optimal way. In the function `ngx_http_proxy_help_store_info_from_header` we were parsing the buffer containing outgoing headers and storing the info from those headers into added help structure `ngx_http_upstream_proxy_t`. By doing so we have wasted memory and also processor time. The information has already been processed as it has been copied to the buffer containing outgoing headers and it has also been already stored somewhere. After deeper analysis of the algorithm evaluating variables from *nginx* configuration we were able to find out how to get values of these variables at the moment of processing the request using internal *nginx* functions. This brought us to the decision of refactoring this patch by removing no longer needed parts of code and using these internal functions instead. The performance benefit of this refactoring is debatable but the real benefit is that we are using already debugged functions which are used at several other places in the code and also that we keep the code clean a readable. Those factors are very important when working with a project as large as *nginx*. Code examples of removed parts follow.

```
static ngx_int_t
ngx_http_proxy_help_store_info_from_header(
        ngx_http_request_t *r,
        size_t len,
        u_char *data)
{
  char *str, *substr, *cr, *lf;
  size_t sublen;
  char *host_str = "Host: ";

  str = ngx_palloc(r->pool, len + 1);
  if(!str) {
    return NGX_ERROR;
  }

  ngx_memcpy(str, data, len);
  str[len] = '\0';
  substr = strstr(str, host_str);

  if (substr) {
    substr += strlen(host_str);
    cr = strchr(substr, CR);
    lf = strchr(substr, LF);
    sublen = (cr < lf ? cr : lf ) - substr;
    r->upstream->proxy_info.host.len = sublen;
    r->upstream->proxy_info.host.data =
        ngx_palloc(r->pool, sublen);

    if (!r->upstream->proxy_info.host.data) {
      return NGX_ERROR;
    }

    ngx_memcpy(r->upstream->proxy_info.host.data,
               substr,
               sublen);
  }

  substr = str;
  cr = strchr(substr, ' ');

  if(cr) {
    sublen = cr - substr;
```

```
    r->upstream->proxy_info.method.len = sublen;
    r->upstream->proxy_info.method.data =
        ngx_palloc(r->pool, sublen);

    if (!r->upstream->proxy_info.host.data) {
      return NGX_ERROR;
    }

    ngx_memcpy(r->upstream->proxy_info.method.data,
               substr,
               sublen);
  }

  return NGX_OK;
}
```

Listing 3.1: ngx_http_proxy_help_store_info_from_header

```
typedef struct {
  ngx_str_t                     host;
  ngx_str_t                     method;
} ngx_http_upstream_proxy_t;
```

Listing 3.2: ngx_http_upstream_proxy_t

Chapter **4**

Testing, comparison, documentation

This chapter focuses on work that is necessarily connected with every software development although this work is not visible to anyone else then the developer. By this we mean testing, documentation and measuring.

Here we will take a look at the development of our testing environment and shortly describe the process the application has to go through to make it to the production. Then we focus on comparing the performance of our improved version with the original one and make a note about stability testing. In the end of this chapter we are going to take a look at the documentation and briefly describe code maintenance pattern we were using.

4.1 Testing and production environment deployment

4.1.1 AMAST - Advanced Measuring and Stress Testing

First we want to take a look at our newly created testing environment *AMAST*. It was developed just for the testing our *nginx* patches therefore its source stays closed and remains covered by the license agreement. We will however describe its functionality and it will provide us with testing results which we will later take a look at.

AMAST is written in Python 3 and it uses modular architecture, just like *nginx* . It consists of client side and server side scripts. It gets launched at the client side while *nginx* is being run on the server side. Its core ensures running requested version of *nginx*'s binary at the server side. This means it cleans the cache directories before the tests are started, launches *nginx* with

requested configuration and measures performance parameters during tests. After that it parallelly starts the modules which represent the actual tests. At the end of the testing it terminates *nginx*, wipes temporary files and presents results to the user.

4.1.1.1 Parallelism

AMAST simulates multiple users by forking a process for each one of them. The number of users is given as a command line argument. These processes then launch chosen testing modules and run the tests. At its current implementation modules have to be written in Python but this is going to change. Modules need to be prepared for being launched in parallel but have the option to alter this behavior and run sequentially.

4.1.1.2 Modules

Modules represent the actual tests that a user can launch. Each module needs to follow a predefined structure which gives it the possibility of modifying the environment on tested server, running the test itself and then manipulating the results to make them human readable. We are now going to describe the implemented modules and what do they test.

- **Range request module** This module tests the functionality of the range request patch. Each user creates approximately 100 range requests and sequentially sends them to the tested server. After sending the first request which was supposed to cause caching, it checks if the file is actually in cache. It does this by configuring *nginx* to insert a special header containing the cache status of the requested file and comparing this value with predefined strings. It also checks the md5 hash of the received data to see if everything arrived as it was supposed to. This module has returned zero errors when run against the final implementation of our patch and therefore we can say our patch is fully functional. This has been checked by capturing network traffic between client, *nginx* and origin using Wireshark. This has confirmed the results of *AMAST*. Therefore we have deployed the patched version of *nginx* into production where it is currently running without any issues.
- **Cache statistics module** This module runs a set of sub-tests as there are more things to test. At first it checks that *nginx* is correctly reporting the number of files and their size on the disk. Then it restarts *nginx* and checks that the statistics have been correctly loaded into the memory. After that it fills the cache with random files, sets their lifetime in cache to a very low number and repeatedly sends requests to some of them. Those which are not requested during their lifetime get deleted and this sub-test checks that our patch gets informed about this and keeps

displaying correct data. Another sub-test sends special *purge* requests to *nginx* which delete the file from cache without taking its lifetime or validity into account. It then checks that cache statistics reflect this situation. Finally, our last sub-test fills the cache with more data then its size and checks that even if files were deleted this way our patch still behaves correctly. As expected, all tests returned zero errors and this patch got included into the code which runs in production. Today, we are successfully parsing this data and then create reports for customers and set their billing plans based on it.

- **Simple request module** This module is only supposed to confirm that a file got cached and that *nginx* returns correct data. Therefore it checks md5 hashes of received data and also the values of previously specified headers containing cache status of requested file. It is not a surprise that this module as well as others does not return any errors when run against our patched version.

There is no module for testing the unique upstream download patch as this gets tested by the simple request module and also by the range request module as they request the same file multiple times. Checking the number of concurrent connections from *nginx* to origin has been done by running *nginx* inside of a tool called *strace* and looking for *connect(2)* calls. This has verified our presumptions that the patch is performing correctly.

4.1.2 Real life everyday usage

Although this part is not required we will take a brief look at it as not every application has the possibility to be deployed to production servers and used in real life. Our patched version of *nginx* runs on several powerful servers and provides worldwide service. We will shortly mention a few things behind the scene that are required apart from patching and developing *nginx*.

4.1.2.1 Deployment system

As we are constantly developing new patches we deploy new versions very often. This has created a need for a deployment system. This system is currently in development and expected to be done before the end of July 2015. What we want to achieve is full automation of the process of compiling the sources, testing the binary, building a package and installing this package to a chosen group of servers. Our deployment system will be connected with our git repository and in the event of pushing to particular branch it will download a copy of this branch's last commit, compile the sources with predefined compiling options and run *AMAST* to test compiled binary. A result of these tests will be displayed to the user and the user will choose between running the tests

again, creating a package or discarding this version. When a choice to create a package will be made the user will have the option to specify the servers this package will be deployed to. Then the package will be deployed and *nginx*'s binary reloaded.

4.1.2.2 Monitoring

Our advanced monitoring system is a must when we want to be able to react appropriately to unexpected situations. It uses some third party applications but mostly it was written from the scratch. It uses small scripts called "sensors" that run on our production servers. These scripts do all kinds of checks like parsing log files, pinging, measuring latency, checking DNS records and so on. They also periodically send data to our database servers where they get processed. Data from those databases are then processed by our notification system and if an error occurs we receive numerous emails, messages and sometimes even calls. In case something stops working we usually know about this situation within seconds and are able to start solving it.

4.2 Comparison of original and patched version

4.2.1 Performance and stability measuring

When we were designing *AMAST* we already had in mind that we will not only need to test the functionality of our patches but also to measure the overall performance of patched *nginx*. Therefore, we have developed it in a way that it is able to measure important connection times. As a result we were able to use a single tool to test that our patches are working and that there was a performance increase. For measuring stability we have also implemented an option to run *nginx* through *valgrind* - a powerful tool which provides features for detecting memory issues in programs.

4.2.1.1 Using AMAST and valgrind together

In the early stages of the testing our patches we were always manually checking *nginx* error logs to see if any errors arise. When developing more patches at once and during the testing of many *nginx* versions we have realized this activity has to be automatized.

Therefore, we added an option to run *nginx* inside of *valgrind* and then copy its output over the network. This required slight changes in *AMAST* but gave us the possibility to test functionality & stability at the same time. It is possible that our patches are working but also contain programming bugs which result in stability decrease. Such bugs are usually difficult to find as we are not able to run *nginx* inside of a debugger or memory checking tool in

the production environment. By connecting *AMAST* and *valgrind* together we have created a possibility to simulate user behavior and see if that reveals any programming failures.

As a result we have fixed numerous hidden bugs before bringing the application to the production and now we have an absolutely stable and patched version of *nginx*. We also created a tool which can be quickly used to test newly compiled binaries with just one command. When the binary passes the tests we can be sure that after deploying such a binary our network will be fully functional.

4.2.2 Comparing important performance markers

After making sure that our version is functional with our patches and we have done everything we could to eliminate hidden bugs we also had to make sure there was a performance increase after applying those patches. This was also obvious at the time of designing *AMAST* therefore we will also use it for this.

4.2.2.1 Connection times

The most important connection marker is time. Whenever the newly patched version replies faster then the original one we have successfully increased the performance of *nginx*. The time the client will have to wait for its data is the only real thing the client can see and so it is also the priority to focus at. To make sure this is happening we have designed *AMAST* to be able to measure two important times: Time-To-First-Byte (TTFB) and the overall time of the connection. TTFB will probably be the most important figure for the client as that is the time it took to receive first byte of response from *nginx*. Also, the lower this time is going to be the greater is the probability this connection will have its overall time as low as possible. The overall time is there to make sure this is not just a presumption. It is used for revealing bugs during the processing of the request after it has been accepted.

After doing multiple tests and calculating averages of both of these times we can say that our patched version is in several situations faster. When testing normal requests by the simple request module we have discovered that the average TTFB of the unpatched version was 0.1698397 seconds, while with the patched version *nginx* was able to respond in 0.1228479 - this is approximately 1.4 times faster. The overall times were then 1.5392757 and 0.8487641- 1.8 times faster. This is only thanks to the unique upstream download patch. With the range request module, the difference was even greater. Average TTFBs were 0.0932075 and 0.0071537 - our patched implementation is therefore approximately 14 times faster then the original one. Overall time was approximately 9 times faster - 0.3148339 versus 0.0352955. This is the per-

formance increase provided by the combination of unique upstream download patch and the range request patch. We can say that we have not noticed a single situation where these times are worse than times of the original implementation. This proves success of our improvements.

4.3 Documentation and code maintenance

4.3.1 Documentation

As required, we have created a documentation for all parts of this thesis. Documentation of implemented improvements consists of main idea description, algorithm description and changes in files. It also includes important warnings for patches that might require refactoring. Documentation of tests consists of their description and usage examples. Documentation of comparison results consists of *AMAST* outputs and their description.

4.3.1.1 Implemented improvements

All implemented patches are successfully working and deployed in production. Documentation of improvements is covered by the license agreement and therefore is not publicly available.

4.3.1.2 Tests

Our newly created testing environment has successfully tested all our patches. Documentation of tests is covered by the license agreement and therefore is not publicly available.

4.3.1.3 Comparison results

Comparison of important connection times has shown our implementation is performing better then the original one. Comparison results can be found on the enclosed CD in the directory `results`, their documentation is however covered by the license agreement and therefore is not publicly available.

4.3.2 Code maintenance

Also a short note on code maintenance is going to have its place here. As a version control system we use *git*, with our repositories being stored in cloud storage *www.bitbucket.com*. We maintain 3 permanent branches - one master branch for a stable code which runs on production servers, one development branch for adding new features and one branch with original sources of *nginx*.

If a serious bug is found in a stable version we checkout a new branch from master, develop a hotfix and then merge these two branches back together.

Features are usually added by checking out a branch based on development branch and then merging them back into it. When a new version of *nginx* gets released we commit that to the originals branch and then merge this branch into development. That way we do not have to apply our patches every time new *nginx* version gets out. And once the version from development branch gets tested it is merged into master branch and a package is created.

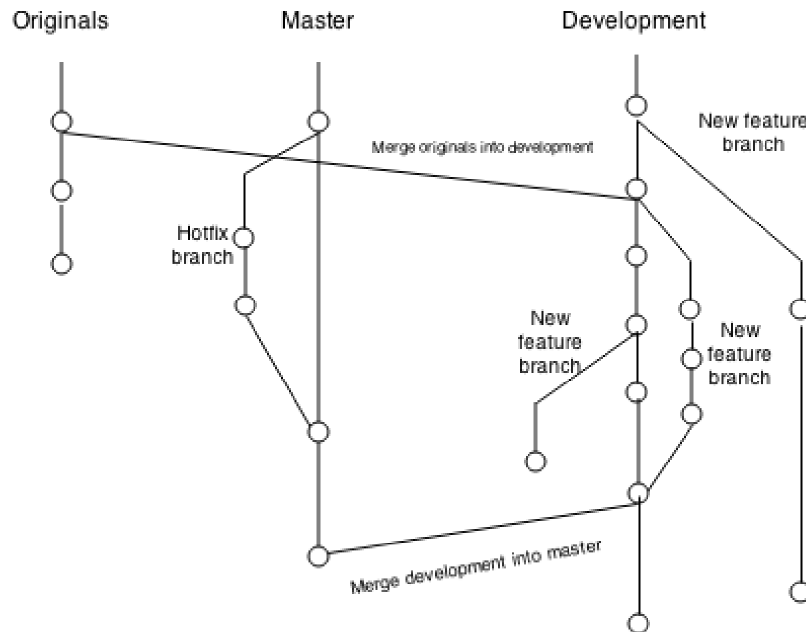


Figure 4.1: git repository layout

Chapter **5**

Outro

5.1 Successes and failures until today

So far we have implemented several improvements into *nginx*, developed a working testing environment and successfully deployed our *nginx* version. This version now runs at approximately 25 servers worldwide and powers the network of one content caching network provider who wished to stay anonymous. At the time of writing this thesis our servers handle more then 50 gigabits of traffic per second and all of this is processed using our patched version of *nginx*. As for our failures, we have not encountered a single problem we were not able to solve during the design & development of these patches. However, our network had a short outage recently but fortunately it was not the fault of *nginx*.

5.2 Plans for the near future

Currently we are at the end of development of another patch which will provide us with the ability to split the cache among multiple drives. *The idea behind this patch as well as the benefits coming from it are both well described in chapter 3, section 3.1.2.* The patch is now implemented and it is in the last part of testing - it has been deployed to production servers with lower then average traffic. In addition, we are currently preparing a first version of Internet Draft document describing the range request patch. We plan to send this document to IETF and see if it becomes an RFC.

As for the live development, we are now working on the implementation of speed parameter for mp4 video files, *nginx* deployment system, refactoring of *AMAST* and other ideas. The speed parameter will allow customers to set maximum transfer rate, deployment system is going to prevent mistakes & save time and we have discovered unnecessarily repeated parts of *AMAST*.

Conclusion

At the end of our work we can state that we have successfully achieved the aim of this thesis and not only that. As well as implementing a performance upgrade to the caching mechanism which is represented by the range request patch and a unique upstream download patch, we have also added several usability patches. In particular the cache statistics patch which added a feature to see information about files in cache per virtual server and then the upstream log patch which allowed us to log the communication between *nginx* and upstream origin server.

Before the implementation of these patches we managed to get advanced level understanding of used algorithms & data structures by studying the source code. This helped us with the implementation and also unified gathered knowledge into one document. After the implementation we were able to create a testing environment called *AMAST* which has proven that our patches are working, stable and well performing. Thanks to the way we implemented *AMAST* we were able to use it in cooperation with a memory checking tool *valgrind*. Therefore we used one single tool for testing functionality, stability and performance. Even though that was enough to fulfill the requirements of this thesis, we also deployed our application in real production environment and let it serve thousands of requests per second. It is currently still running without anyone's intentions to be replaced. In the end we have also created the documentation of our improvements, tests and comparison results. This can be partially found on the enclosed CD.

Everything summed together obviously means we have fulfilled the goal of this thesis beyond the point we were able to imagine at its beginning. During the time of creating this thesis we gained significant knowledge, improved our programming skills and deployed our application into production.

Bibliography

[1] V., J. *Forward Proxy vs Reverse Proxy [online]*. JSCAPE, Aug 2012, [cit. 2015-04-26]. Available from: http://www.jscape.com/blog/bid/87783/Forward-Proxy-vs-Reverse-Proxy

[2] nginx about [online]. [cit. 2015-04-26]. Available from: http://nginx.org/en/

[3] Kernighan, B. W.; Ritchie, D. M. *The C Programming Language*. Prentice Hall PTR, second edition, 1988, ISBN 978-0131103627.

[4] Connection processing methods [online]. [cit. 2015-04-26]. Available from: http://nginx.org/en/docs/events.html

[5] Linux man-pages project. *Linux manual section 2 [online]*. [cit. 2015-04-26]. Available from: http://man7.org/linux/man-pages/dir_section_2.html

[6] Linux man-pages project. *Linux manual section 3 [online]*. [cit. 2015-04-26]. Available from: http://man7.org/linux/man-pages/dir_section_3.html

[7] Miller, E. *Emiller's Guide To Nginx Module Development [online]*. Apr 2007, [cit. 2015-04-26]. Available from: http://www.evanmiller.org/nginx-modules-guide.html

[8] nginx source code [online]. [cit. 2015-04-26]. Available from: http://nginx.org/download/nginx-1.7.12.tar.gz

[9] *Proxy cache path [online]*. [cit. 2015-04-26]. Available from: http://nginx.org/en/docs/http/ngx_http_proxy_module.html

[10] Nottingham, M. *Two Http Caching Extensions [online]*. Dec 2007, [cit. 2015-04-26]. Available from: https://www.mnot.net/blog/2007/12/12/stale

Appendix A

Acronyms

URL Uniform Resource Locator

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

DNS Domain Name System

IP Internet Protocol

RFC Request For Comments

SSD Solid-state drive

AIO Asynchronous I/O

LVM Logical Volume Manager

RAID Redundant Array of Inexpensive Disks

ZFS Z File System

AMAST Advanced Measuring and Stress Testing

OS Operating System

API Application Programming Interface

IDE Integrated Development Environment

URI Uniform Resource Identifier

CGI Common Gateway Interface

TCP Transmission Control Protocol

CPU Central Processing Unit

SSL Secure Sockets Layer

CSS Cascading Style Sheets

RAM Random Access Memory

I/O Input/Output

LRU Least Recently Used

TB Terabyte

CD Compact Disc

Appendix B

Contents of enclosed CD

```
src ............................ the thesis LaTeX source codes directory
  BP_kvasnicka_tomas_2014.tex ........ the thesis text in TEX format
text ........................................ the thesis text directory
  thesis.pdf .......................... the thesis text in PDF format
results .......................... the directory of comparison results
  normal_request .............. the directory of normal request results
  normal_request ................ the directory of range request results
  readme.txt ........... description of computers used for benchmarks
```