



## ASSIGNMENT OF MASTER'S THESIS

<b>Title:</b>	Suitability analysis of Kubernetes for Seznam.cz
<b>Student:</b>	Bc. Ondřej Šejvl
<b>Supervisor:</b>	Ing. Jan Baier
<b>Study Programme:</b>	Informatics
<b>Study Branch:</b>	Web and Software Engineering
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	Until the end of summer semester 2016/17

### Instructions

Make yourself familiar with the Kubernetes system. Compare Kubernetes in different environments (filesystem, network management, ...) and create a set of recommendations for Seznam.cz corporate machines. Design and implement a set of applications to simulate real traffic. Describe how Kubernetes can help developers to deploy a software in production environment.

The applications should fulfill the following requirements:

- Load configuration files, SSL certificates and static resources.
- Use persistent storage.
- Raise controlled faults, deadlocks, segmentation faults.
- Simulate heavy CPU and RAM load.
- Log their own activity and reliably transfer the logs to the central log storage.

Implement application servers ready for Kubernetes in Python and Go and run a set of benchmarks on them.

### References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.  
Head of Department

prof. Ing. Pavel Tvrdík, CSc.  
Dean

Prague January 23, 2016



CZECH TECHNICAL UNIVERSITY IN PRAGUE  
FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF SOFTWARE ENGINEERING



Master's thesis

**Suitability analysis of Kubernetes  
for Seznam.cz**

*Bc. Ondřej Šejvl*

Supervisor: Ing. Jan Baier

10th May 2016



---

# Acknowledgements

Most of all I would like to thank to my thesis advisor Ing. Jan Baier for his expert guidance and invaluable remarks. I would also like to thank to Ing. Tomáš Kukrál for his counsel during my work and for a lot of open source material.

To the Seznam.cz company and to my team leader David Bouček in particular, I thank for providing me with hardware, introducing me to technologies like Kubernetes and Docker, letting me dedicate a part of my working time to this thesis and constantly bringing in fresh ideas for improvements.

My thanks belong to Martin Stružský for the proofreading as well.

And of course, this thesis wouldn't have been possible without the support of my family, especially my wife.



---

# 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. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 10th May 2016

.....

Czech Technical University in Prague  
Faculty of Information Technology  
© 2016 Ondřej Šejvl. 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**

Šejvl, Ondřej. *Suitability analysis of Kubernetes for Seznam.cz*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016. Also available from: (<https://github.com/sejvlond/master-thesis>).



---

# Abstrakt

Tato práce analyzuje použitelnost systému Kubernetes v prostředí společnosti Seznam.cz. V první části je práce zaměřena na teoretické informace a jsou zde identifikovány možné problémy spojené s přechodem na tuto technologii. V druhé části se práce zabývá vyřešením těchto problémů a zprovozněním clusteru s dalšími podpůrnými aplikacemi.

**Klíčová slova** Kubernetes, Seznam.cz, Docker, Heka, Kafka, Linux, virtualizace

---

# Abstract

This thesis analyses the suitability of the Kubernetes system in the Seznam.cz's company environment. The first part of the thesis focuses on the theory and identifies possible problems connected with a switch to this technology. The second part of the thesis describes solving those problems and running the cluster together with other applications.

**Keywords** Kubernetes, Seznam.cz, Docker, Heka, Kafka, Linux, virtualization



---

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Seznam.cz nowadays</b>	<b>3</b>
<b>2 What is Kubernetes?</b>	<b>5</b>
<b>3 What is Docker?</b>	<b>9</b>
3.1 Docker architecture . . . . .	9
<b>4 Kubernetes basic concepts</b>	<b>11</b>
4.1 Pod . . . . .	11
4.2 Volume . . . . .	11
4.3 Replication controller . . . . .	11
4.4 Service . . . . .	11
<b>5 Kubernetes architecture</b>	<b>13</b>
5.1 Etcd . . . . .	13
5.2 API server . . . . .	13
5.3 Kubelet . . . . .	13
5.4 Kube-proxy . . . . .	13
5.5 Controller manager . . . . .	13
5.6 Scheduler . . . . .	14
5.7 Node . . . . .	14
<b>6 Possible problems with Kubernetes in Seznam.cz</b>	<b>17</b>
6.1 Docker registry . . . . .	17
6.2 Secrets distribution . . . . .	17
6.3 Logging . . . . .	18
6.4 Security . . . . .	18
6.5 Monitoring . . . . .	18

6.6	Static content of websites . . . . .	18
<b>7</b>	<b>Running Kubernetes</b>	<b>21</b>
7.1	Networking in Kubernetes . . . . .	21
7.2	Starting cluster . . . . .	24
<b>8</b>	<b>Docker registry</b>	<b>27</b>
<b>9</b>	<b>Secrets distribution</b>	<b>31</b>
<b>10</b>	<b>Logging</b>	<b>33</b>
10.1	FluentD . . . . .	34
10.2	Logstash . . . . .	35
10.3	Heka . . . . .	35
<b>11</b>	<b>Security</b>	<b>41</b>
<b>12</b>	<b>Monitoring</b>	<b>45</b>
<b>13</b>	<b>Static webpages content</b>	<b>47</b>
<b>14</b>	<b>Testing application</b>	<b>49</b>
14.1	Designing Tarsier . . . . .	50
14.2	Tarsiers synchronization . . . . .	51
14.3	Tarsier plugins and their commands . . . . .	52
<b>15</b>	<b>Testing the cluster</b>	<b>55</b>
<b>16</b>	<b>Benchmarking Python and Go application servers</b>	<b>59</b>
<b>17</b>	<b>Benefits for Seznam.cz</b>	<b>63</b>
	<b>Conclusion</b>	<b>65</b>
	<b>Bibliography</b>	<b>67</b>
<b>A</b>	<b>Acronyms</b>	<b>73</b>
<b>B</b>	<b>Contents of enclosed CD</b>	<b>75</b>

---

## List of Figures

2.1	Virtualization based on hypervisor . . . . .	6
2.2	Virtualization based on containers . . . . .	7
3.1	Docker architecture . . . . .	10
5.1	Service overview in Kubernetes . . . . .	14
5.2	Kubernetes services interconnection . . . . .	15
5.3	Kubernetes nodes . . . . .	15
7.1	The path of a packet in flannel network . . . . .	23
7.2	Screenshot of <code>kubectl get nodes</code> output . . . . .	26
8.1	Seznam.cz Docker registry architecture . . . . .	29
10.1	Fluentd . . . . .	34
10.2	Kafkafeeder architecture . . . . .	40
14.1	Wave propagation in Tarsier . . . . .	52
16.1	Benchmarking Python and Go – Requests per second . . . . .	60
16.2	Benchmarking Python and Go – Average response latency . . . . .	61
16.3	Benchmarking Python and Go – Transfer per second . . . . .	61



---

## List of Listings

11.1 Dockerfile snippet . . . . .	42
14.1 Tarsier plugin's interfaces . . . . .	50
14.2 The <code>heavy_load/spin_cpu</code> command body structure . . . . .	51
14.3 The <code>persistent_storage/write</code> command body structure . . . . .	51
15.1 Snippet of replication controller configuration . . . . .	57





---

# Introduction

Given the performance of today's servers, it is almost impossible to use their full potential with a single application. That's where the possibility to install more than one application on each server comes to mind. However, this technique poses certain risks. What about when one application needs specific libraries or a whole environment in a certain version and another one needs something different? That's exactly when virtualization fits. Using virtualization we can run many applications on a single machine without them knowing about each other.

Virtualization does not solve all problems and issues. One of the main problems with virtualization servers is a very difficult scaling as there is no easy way to automatically react on application's needs. Simple example: Let's have a website and create a virtual environment for it which we deploy on 2 servers (because of backup in case of one machine's failure). Together with this application there can be many others running in their virtual spaces. When traffic rises to unexpected levels, the application can quickly demand more resources than the virtual machine can provide and as there are more virtuals on the master, resource allocation cannot be increased. So the whole virtual have to be moved to different machine which has more resources available. And here comes the looking for it. Looking for machine with more resources can be really hard task, because there is also possibility that no other machine have enough resources for this kind of application. So someone have to decided which application can be moved on which machines to make a space for this, actually greedy one. But this situation may occur again few hours later, when this application will not need as much and other one will be busy.

Problems like this – and many others – can be solved using Kubernetes [3]. Kubernetes is an open source system for automating deployment, operation, and scaling of containerized applications. Application and its virtual environment has to be packed in a container. In case of Kubernetes, the recommended container technology is Docker. Kubernetes automatically starts containers

on physical machines in as many instances as the maintainer defines. It is very simple to automatically react on each application's needs, give them more resources, move them among machines, scale them up and down and run auxiliary jobs at night when the flow of many applications is smaller to use the server house more efficiently.

Kubernetes offers all those possibilities and that's why I chose to examine it more closely and start a testing instance of a private cloud based on Kubernetes at Seznam.cz company.

---

# Seznam.cz nowadays

Seznam.cz is a big company with more than a thousand employees. More than one quarter of them are developers. Seznam.cz is divided into compartments and developers in each compartment are grouped to teams led by team leaders. Each team leader is responsible for applications allocated to his team. There are a couple of recommendations on workflow (e.g. unified coding style), but as different teams have different demands and use cases for their applications, they also have a slightly different requirements on coding, building, testing, deployment etc.

All over the company we are using Git as a version control system. Git is installed centrally using GitLab as a management system. GitLab provides CI – Continuous integration [2]. So each team has a possibility to easily run automatic builds or push to a repository after a merge request.

Testing and developing application is each team's responsibility. But when someone wants to deploy an app to a production environment, there is a strict process. Each developer has access rights to the core machine with all the Debian [42] repositories. Developers have to upload Debian packages to the right branch (there are development, unstable, testing and read-only stable branches for each Debian distribution, divided per compartments), and run an incremental build of the branch.

We are using a request tracker system. Developers have to write a request consisting of packages which have to be moved to the stable branch, steps to install and run them on each server and list of servers affected by this change. A process to rollback has to be described too.

An administrator from the team which manages the affected servers then takes the ticket and starts deploying. He can consult the process with the developer, who then tests the application to check whether the deployment was successful and then traffic is renewed (if it was stopped before) and the installation continues on the next server in the list.

This means developers have to turn anything they want to deploy to Debian packages, which depend on other packages. It is quite a simple but an

efficient way to manage dependencies, to list software installed on each server or to see who is responsible for changes and maintenance of the package.

The problem with this process arises when you want to use anything that does not have a Debian package. Developers then have to create one from an upstream source code (that is not a problem) and maintain it so that it doesn't become obsolete after a while – and that is problem.

The same situation can happen when developer wants to use a package from pip [56] or an updated version of a package, that wasn't updated in the official Debian repository. Adding such packages to the tree leads to a massive growth of the repository and a number of patched versions of software.

---

## What is Kubernetes?

Kubernetes is an open source platform for automatic deployment, scaling and operation of application containers in a cluster [3]. The main goal of Kubernetes is to manage the ecosystem of components in a custom public or private cloud. The emphasis is on high availability with scaling as an essential element.

Scaling needs to be taken into account during the development of any kind of a high availability application. The problem is when the application cannot fully use the performance of the physical server. Because of high availability, the application has to be ran on more machines, which means that the performance of the servers is wasted. The virtualization is the answer. There are many ways of virtualization: from entire operating systems (using VMware [5] or VirtualBox [7]) to container based virtualization (such as widely used OpenVZ [6]).

All those technologies allow to run more virtual applications on a single physical machine. It provides better load balance and server efficiency while preserving isolated environment for each application spread over multiple machines.

The problem with this kind of virtualization arises when larger traffic comes to an application or more performance and resources are requested: there is no easy way to react accordingly. The migration of a whole container can also be quite difficult as was written in the previous section.

Another type of virtualization focuses on simplifying virtualization procedures and allowing prompt reaction on application's needs. Examples of such virtualization technique are technologies like Docker [11], LXC [8] and others. The virtualized environments are packed in small containers where no hypervisor is needed, which reduces virtualization complexity and increases speed of deployment, starting, scaling etc.

Kubernetes uses such containers, so applications are separated from the internals and can be easily moved among machines. Those containers work only with logical resources that Kubernetes provides them. Containers can

## 2. WHAT IS KUBERNETES?

---

be built and deployed automatically and as often as needed, which allows continuous deployment and easy delivery. Thanks to containers, applications can be separated to small pieces and used as micro-services. And last but not least the same containers can be used in test, staging and production environments. The only thing that is changing is Kubernetes configuration and environment which allows very easy development and testing.

Kubernetes is not a PaaS (Platform as a Service) solution because it does not limit the type of applications and does not dictate what application frameworks, languages or runtime libraries have to be used.

Kubernetes supports Docker and Rocket [9] containers at the moment. We will examine Docker more closely in the following chapters.

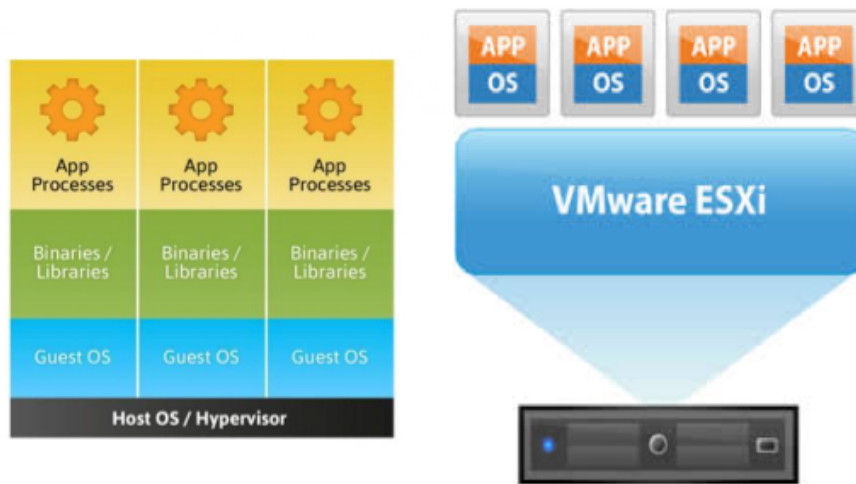


Figure 2.1: Virtualization based on hypervisor [10]

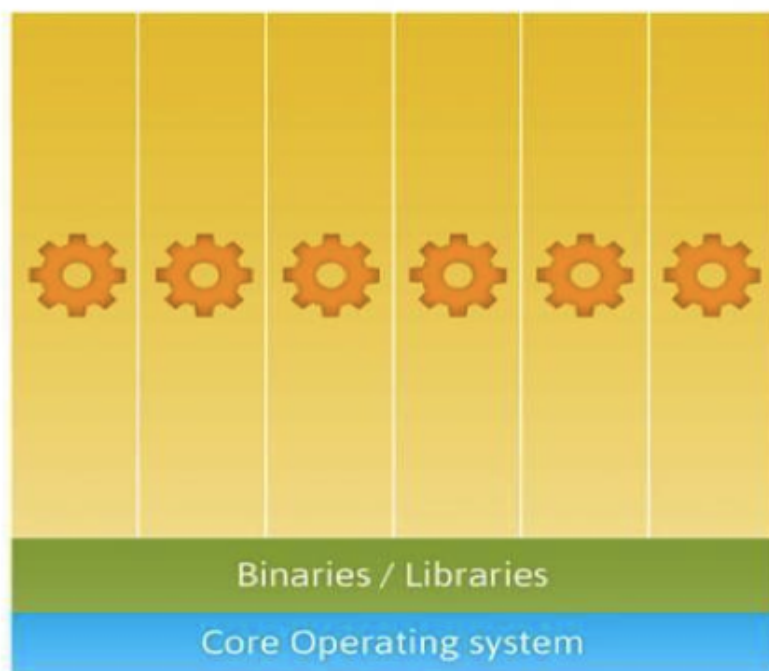


Figure 2.2: Virtualization based on containers [10]





---

# What is Docker?

Docker is an open source platform for developing, deploying and running applications. Docker allows to separate applications from infrastructure using environment virtualization, which is empowered by virtualization support in kernel and by tools that help to manage and deploy applications. Containers are used again, but they are not as independent as OpenVZ containers for example. Docker uses containerization support in kernel, so applications share kernel with the host machine. This makes containers more lightweight, faster to run and portable. Docker allows more advantages thanks to it is wrapping basic kernel virtualization

## 3.1 Docker architecture

Docker daemon runs on the host machine and manages all Docker images and containers. Docker client communicates with this daemon to create images and to upload them on Docker registry. The figure 3.1 shows the whole concept of it.

### 3.1.1 Docker image

Docker image is a read-only template. It contains a chosen operating system and a set of installed applications. Images used to run simple binary program can also start from scratch.

### 3.1.2 Docker container

Docker container is a structure holding everything what application may need to run. Each container is based on an image. There can be multiple containers based on the same image, all of them can be started, stopped, moved or deleted individually and users can attach to any of them. Containers provide an isolated and a secure platform for applications.

### 3. WHAT IS DOCKER?

---

#### 3.1.3 Docker registry

Docker registry is a server-side application holding images. It is a storage, where clients can push their images so anyone can pull them and run containers based on them.

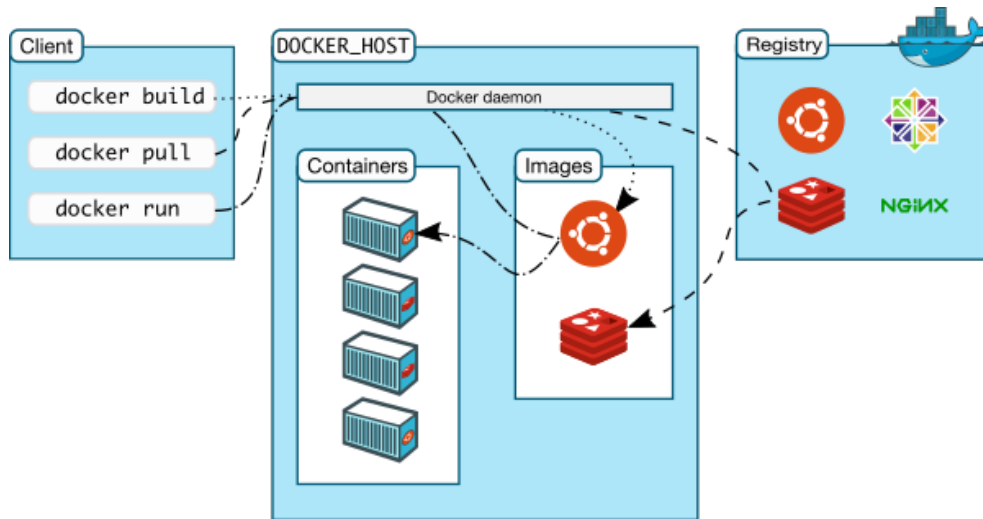


Figure 3.1: Docker architecture [12]

---

# Kubernetes basic concepts

## 4.1 Pod

In Kubernetes all containers are run in so called pods. Pod is a package of more containers (or just one) which logically belong to each other (for example application server and its proxy). It is against Docker principle to run more processes in a single container, but proxy and server have to be on same server so they can communicate via localhost and don't overload the network. Pod is the basic building block so it is provided that containers belonging to it will be run together on the same machine. Scaling is done with whole pods.

## 4.2 Volume

In Docker a volume is a directory on a filesystem or in another container. Kubernetes volumes are more abstract. Kubernetes volume has a defined service life, which is the same as the service life of the pod which requested it. Pod may contain none or more volumes and each container in it has to define where the volume should be mounted.

## 4.3 Replication controller

Replication controller (RC) specifies the amount of pods to run at the same time. If there is fewer, it creates new ones, if there is too many, it stops some. RC keeps running exactly the same instances of pods as the maintainer wants to.

## 4.4 Service

Pods in Kubernetes may start and quit as the replication controller settle. Each pod has its own IP address which can change during time so it is better

#### 4. KUBERNETES BASIC CONCEPTS

---

not to rely on it, because pod can be moved to another machine. Which leads to a problem: what if a pod (and container in it) needs to communicate with another one? Here comes the Kubernetes service. It is an abstraction layer which defines a logical set of pods and rules for accessing them. Sometimes it is called a micro-service as well. As an example I can mention a backend service which is running in 3 instances. The frontend does not care about which one of them it will be communicating with.

---

# Kubernetes architecture

## 5.1 Etcd

Etcd is a distributed, consistent key-value store for shared configuration and service discovery [14].

## 5.2 API server

The Kubernetes API server validates and configures data for the API objects which include pods, services, replication controllers, and others. The API Server services REST operations and provides the frontend to the cluster shared state through which all other components interact. [4]

## 5.3 Kubelet

Another very important part of Kubernetes is Kubelet. Kubelet is an agent running on every node and it provides starting and stopping containers.

## 5.4 Kube-proxy

Every node in Kubernetes cluster has its own kube-proxy. This application watches the Kubernetes master node and in case of adding or deleting a service it will open or close ports (even randomly chosen) on the local node. Each connection is then forwarded to the right pod.

## 5.5 Controller manager

The Kubernetes controller manager is a daemon that embeds the core control loops shipped with Kubernetes. In robotics and automation applications, a control loop is a non-terminating loop that regulates the state of the system.

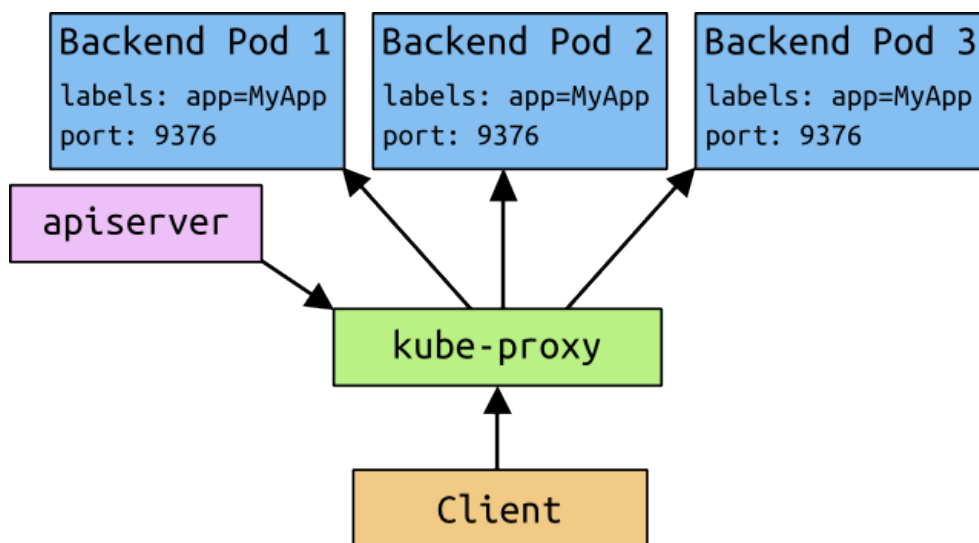


Figure 5.1: Service overview in Kubernetes [4]

In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the API server and makes changes in order to move the current state towards the desired state. Examples of controllers currently shipped with Kubernetes are the replication controller, endpoint controller, namespace controller, and service account controller. [4]

## 5.6 Scheduler

The Kubernetes scheduler runs as a process alongside other master components such as the API server. Its interface to the API server is to watch for pods with an empty `PodSpec.NodeName`, and for each pod, it posts a binding indicating where the pod should be scheduled.

## 5.7 Node

There are two types of nodes in a cluster: the master node and the worker nodes, formerly known as minions. On the master node the API server, the scheduler and the controller manager are running together with etcd and possibly flannel. Each node runs Kubelet and optionally flannel. Using Docker for those main Kubernetes parts, the final structure of the nodes is as follows in the figure 5.3.

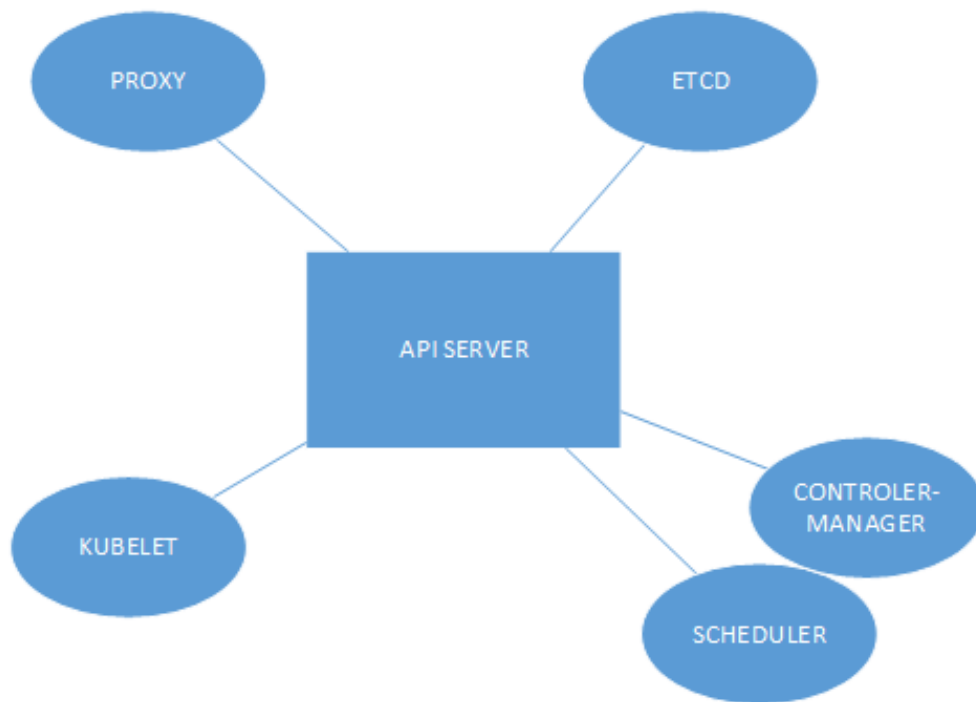


Figure 5.2: Kubernetes services interconnection [13]

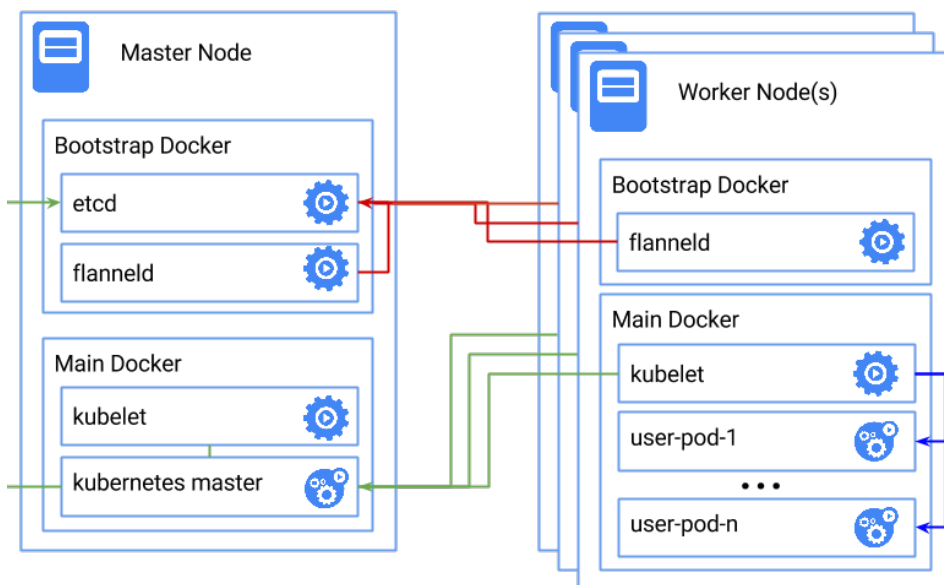


Figure 5.3: Kubernetes nodes [16]





---

# Possible problems with Kubernetes in Seznam.cz

This chapter summarizes what we have to focus on in Seznam.cz so we can build and maintain a Kubernetes cluster.

## 6.1 Docker registry

We have to build our own Docker registry with a support for authentication so it is clear who and when pushed a certain Docker image and who is responsible for it in case of problems. Next we have to run Docker registries for developers both in testing and production environment. Images have to be mirrored from one to another registry. Production registry has to be in each data centre and it has to be a high availability service because if a whole data centre depends on a single server with Docker registry and the server fails (which it will if a whole data centre starts pulling images from it at once), it will be impossible to run new applications.

## 6.2 Secrets distribution

How to share passwords, certificates and other secrets which are different in development and test/production environment and which cannot be available outside of the containers that need them? Moreover, the production secrets are known only to administrators, so there has to be an easy way how to put secrets into containers without compiling them in images. Kubernetes has a technique for it and it calls it Kubernetes secrets. It has to be examined more closely and run in Seznam.cz conditions and environments.

### 6.3 Logging

Logging is a very huge problem. Actually most of our applications are creating debug logs, which are saved on the file system. In case of switching pods among servers there has to be a log collection and their transfer to a central storage and it has to be done so that applications don't get overloaded.

Beyond debug logs we also create so called business logs which are sensitive and cannot be lost because there are more calculations over them.

And there are also third party logs such as access logs from Nginx [37] or Apache [38] and many more. Their collection has to be done as well as gathering server side statistics (CPU load, etc.). There is a Kafka cluster at Seznam.cz for this task, which can also run support jobs to forward the logs to specific databases (Elasticsearch [39], HDFS [40], ...).

Problems may occur with long running applications that generate large log files unless there is a logrotate running. Cron or logrotate [43] do not belong to Docker images, because Docker principle is to run only one application in each container.

### 6.4 Security

At the moment administrators are taking care of host machines and virtual machines running on them and developers only deploy applications in form of Debian packages. When a security issue is discovered (like Heartbleed [41] for example) administrator is able to maintain it and fix whatever it needs on the host machine or in specific virtual machine. When developers will deploy Docker images with operating system inside, fixing such images in case of security issue needs to be solved. Definitely we cannot rely that developers will rebuilt images in a few minutes. In case of automatically built images the whole environment for it has to be built and we also the authorization have to be solved sufficiently.

### 6.5 Monitoring

We need to monitor nodes in cluster and also applications and containers in Kubernetes. We are testing Prometheus [29] for metrics collection, while monitoring what is running where is built in Kubernetes itself.

### 6.6 Static content of websites

When deploying a website, there are always at least 2 versions running at the moment. The old one, where most of the traffic is going to, and the new one. The problem comes when user sends a request and the new version responds. User will receive a HTML code with links to JavaScript and CSS files and

send a request for these files to the data centre. The load balancer and other services on the way may point this request to the pod with the old version of the website so none or wrong files will be downloaded and user will see the page with errors (or nothing at all).

At the moment we are deploying static content of websites first and it is installed on all machines. Static content is versioned in its path, so requests for it are always successful. Than one machine is removed from the pool (and waits until all currently opened connections are done) and it is replaced with a new version. This ensures that downloading static content never fails. However, this procedure will no longer be available in Kubernetes.

These are the problems which I have to find solution for.



---

# Running Kubernetes

I will run Kubernetes on 3 testing machines. First decision I have to make is which operating system I will use. Since almost all machines in Seznam.cz data centres are running Debian I will use it as well on my testing cluster and I will install its newest stable version Jessie. There are Linux distributions made especially for Kubernetes like Fedora Atomic [44], but as our administrators have many years of experience with Debian, it will be better to use it instead of changing architecture to cloud and changing Linux distribution at the same time.

Another decision that has to be made is about networking.

## 7.1 Networking in Kubernetes

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. They give every pod its own IP address so I do not need to explicitly create links between pods. This creates a clean, backwards-compatible model where pods can be treated much like VMs or physical hosts from the perspective of port allocation, naming, service discovery, load balancing, application configuration, and migration. [4]

### 7.1.1 Docker model

Before discussing the Kubernetes approach to networking, it is worthwhile to review the “normal” way that networking works with Docker. By default, Docker uses host-private networking. It creates a virtual bridge, called `docker0` by default, and allocates a subnet from one of the private address blocks defined in RFC1918 [45] for that bridge. For each container that Docker creates, it allocates a virtual Ethernet device (called `veth`) which is attached to the bridge. The `veth` is mapped to appear as `eth0` in the container, using Linux namespaces. The in-container `eth0` interface is given an IP address from the bridge’s address range.

The result is that Docker containers can talk to other containers only if they are on the same machine (and thus the same virtual bridge). Containers on different machines cannot reach each other — in fact they may end up with the exact same network ranges and IP addresses.

In order for Docker containers to communicate across nodes, they must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or else be allocated ports dynamically. [15]

### 7.1.2 Kubernetes model

Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control. Dynamic port allocation brings a lot of complications to the system – every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- all containers can communicate with all other containers without NAT
- all nodes can communicate with all containers (and vice-versa) without NAT
- the IP that a container sees itself as is the same IP that others see it as

In reality, Kubernetes applies IP addresses at the pod scope – containers within a pod share their network namespaces – including their IP address. This means that containers within a pod can all reach each other's ports on localhost.

This networking mode is implemented in many different ways. The basic one and the one which is mentioned in documentation of Kubernetes is flannel. [15]

#### 7.1.2.1 Flannel

Flannel is a virtual network that gives each host a subnet for use with container runtimes.

Platforms like Google's Kubernetes assume that each container (pod) has a unique, routable IP inside the cluster. The advantage of this model is that it reduces the complexity of doing port mapping.

Flannel runs an agent, `flanneld`, on each host and is responsible for allocating a subnet lease out of a preconfigured address space. Flannel uses `etcd` to

store the network configuration, allocated subnets, and auxiliary data (such as hosts' IP addresses). The forwarding of packets is achieved using one of several strategies that are known as backends. The simplest backend is UDP and uses a TUN device to encapsulate every IP fragment in a UDP packet, forming an overlay network. The following diagram 7.1 demonstrates the path a packet takes as it traverses the overlay network. [1]

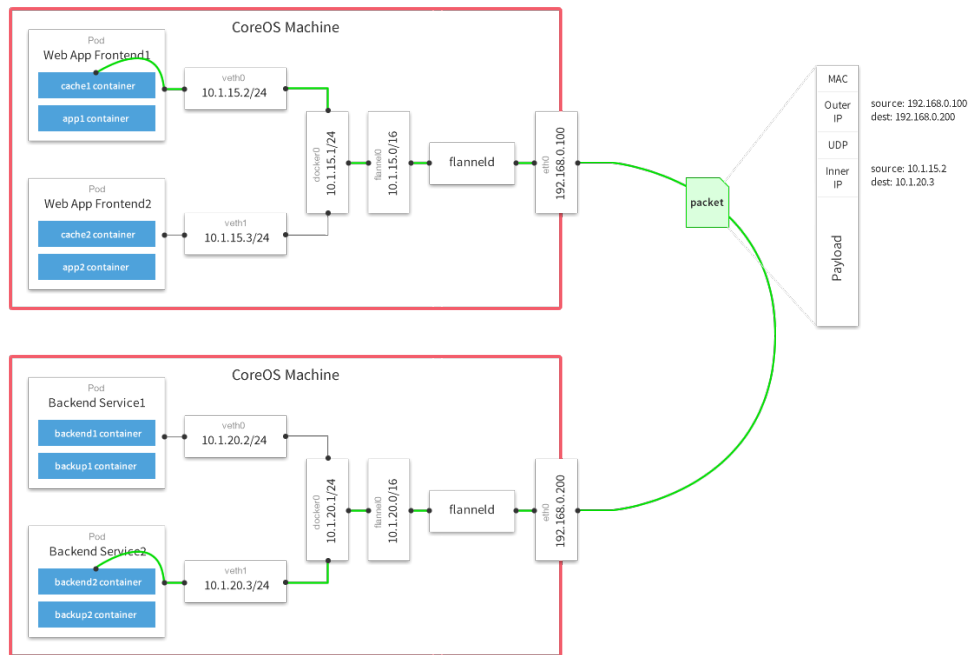


Figure 7.1: The path of a packet in flannel network [1]

Flannel is simple to start up but there can be some overhead expected. It is needed to test how big overhead it will be. Another big disadvantage of flannel is that it is not multitenant and so there is no way how to define any rules who can communicate with whom and who can't. In practice that means everyone sees everyone. For introducing cloud at Seznam.cz it can be sufficient but in the future this needs to be solved more properly so there can be policies defining restricted access to sensitive services.

### 7.1.2.2 Calico

Next networking option is Calico. Calico provides a highly scalable networking solution for connecting data center workloads (containers, VMs, or bare metal). It is based on the same scalable IP networking principles as the internet: providing connectivity using a pure Layer 3 approach. Calico can be deployed without encapsulation or overlays to provide high performance at massive scales.

When using Calico networking in containerized environments, each container gets its own IP address and fine grain security policy. A calico-node service runs on each node which handles all of the necessary IP routing, installation of policy rules, and distribution of routes across the cluster of nodes. [17]

Calico has even a section dedicated for Kubernetes in their manual [18]. They state that thanks to that there is no overlay, Calico will be faster than technologies that use overlay, such as flannel. Calico is using the Bird [19] system for route distribution around the network.

### 7.1.2.3 OpenContrail

On the meeting with company tcp cloud [46] technology OpenContrail was discussed [20]. OpenContrail is a network virtualization platform for the cloud. It has been designed with scale out in mind [21]. OpenContrail is a representative of SDN (software defined networking) and it offers to define custom policies of containers communication thanks to label system in Kubernetes. It will be worth it to examine whether this technology fits for Seznam.cz needs and environment.

## 7.2 Starting cluster

From listed options of network management I decided to start with a simple one: the flannel. This thesis should create a proof of concept that it is possible to maintain Kubernetes in Seznam.cz. I want to create a testing application and an example Kubernetes cluster where I solve all potential problem described in the previous chapter and then I give this to our administrators who may test it further as they want to.

Running flannel containers needs privileged permissions but user defined pods and their containers should not ever have such permissions. So the best way how to achieve this behaviour is to start two separate Docker daemons where one will allow to create privileged containers and the second one won't.

Starting Docker on Seznam.cz corporate machines brought a couple of problems that must be solved first. From Seznam.cz system preinstaller in `/etc/network/interfaces` all routes to private IPs are routed via `eth0` interface which cases that Docker could not find and private IP range free for its purposes. This can be simply solved by freeing an IP range and restarting network service together with Docker daemon.

Next issue that occurred was a little bit tougher. In the Docker log I found that no chain exists for an iptables rule which Docker wants to set. After consulting this problem with my team leader it proved to be caused by missing kernel modul `xt_conntrack`. After adding this module to kernel Docker daemon finally started.



In the Docker log I also found a warning that cgroup memory is not allowed and that could possibly cause Kubernetes to not work properly with pod memory limitation. I added the following line to the `/etc/default/grub` file and updated grub.

```
1 GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

With the Docker daemon running I could start Kubernetes. As was shown in the Kubernetes basic concepts chapter, the master node has to have API server, scheduler, proxy, kubelet, controller-manager and flannel running in a separate Docker daemon.

The first step is to create a bootstrapped daemon. The second is to run `etcd`. I used `gcr.io/google_containers/etcd-amd64` image. The third is to setup flannel. I also used a prebuilt image, this time `quay.io/coreos/flannel`. Finally the last step is to start kubelet. Kubernetes offers prebuilt Docker image with all its components included called `gcr.io/google_containers/hyperkube-amd64`. From the hyperkube image the kubelet is called and in configuration of the kubelet there is a set of static pods that should be created. The hyperkube firstly creates those pods and then starts the kubelet. Those pods are the scheduler, the API server, the controller-manager and the kube-proxy.

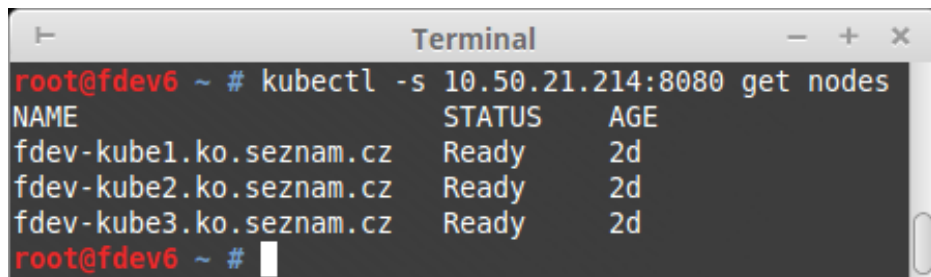
I have prepared a script where this whole configuration is scripted and which may be called with the following single line: `./kube.sh start-master` You can also specify which components should be started.

Now the worker has to be set. The configuration is similar only there is not so many components to run. The first step is also to create a bootstrapped daemon, then the flannel and the kubelet are started, respectively. The kubelet is configured to run only kube-proxy.

Now adding a new worker node to the cluster is very simple. Install the system, fix problems described earlier and then upload a script and run `./kube.sh start-worker`. This system may be automated with ansible [47] for example but this is a task for our administrators.

Also there are other possibilities how to run this Kubernetes model. The kubelet can be installed on bare-metal and not be dockerized and the same applies to the flannel. But it really is just a matter of personal preferences.

The cluster is now fully functional and pods may start to be distributed.



```
root@fdev6 ~ # kubectl -s 10.50.21.214:8080 get nodes
NAME                                STATUS    AGE
fdev-kube1.ko.seznam.cz             Ready    2d
fdev-kube2.ko.seznam.cz             Ready    2d
fdev-kube3.ko.seznam.cz             Ready    2d
root@fdev6 ~ #
```

Figure 7.2: Screenshot of kubectl get nodes output

---

## Docker registry

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. [22]

The Registry is the only reasonable way how to distribute Docker images inside a network. When a developer creates new image with his application, there are 2 ways how to get it to production. The first one is to do it through a Dockerfile. The second one is using Docker image. In the future it would be nice to have a whole system, where you only put a new Dockerfile (let's say simply by merging your dev branch with the new Dockerfile to the master branch in Git) and the system will then build your image, tag it properly and upload it to the Docker registry. This is a state we want to achieve eventually, but it currently takes more work than we can afford to build a system like this.

So the second way is that Deznam.cz will have to create a Docker registry, where developers can push their images and then send tickets to administrator to get them in production. So I have to study the Docker registry more deeply. The Registry is an application that you can run on server. Then every developer have to tag his image with server name and port on which the registry is running and append his own name. In Seznam.cz we are prefixing each name of application with the department name and the "szn" prefix. For example `szn-fulltext-APP_NAME`. So basically now each developer will have to tag his image similarly like that: `REGISTRY_URL /fulltext/APP_NAME` and push it. The Registry URL has to be easy to remember and we need to be sure it does not cause conflict in the future. You also have to define which images will be included in the Kubernetes pod and those pod definitions have to be created in a cooperation of developers with administrators. And if there are different names for the registry in development, staging and production environment those pod definitions will have to be updated each time which might easily lead to mistakes.

We have to create development, staging and production Docker registry servers where images will be stored. The development registry can be a stand-

ard server with a large disk space because many versions of images will be stored here. Developers will have unlimited access to it via the Docker command line tool and no special policy will be defined here.

We cannot have the dev registry with authentication because when pushing new image to registry you would have to set password. As the credentials are stored in `~/.dockercfg` and every developer in our development environment has root privileges, such practice would be insecure and untrustworthy.

After successfully creating an image and uploading to the dev server, the developer will have to send a request to an administrator to deploy it. In Seznam.cz we have a custom request tracker and when someone wants to deploy something, we have to send a RT ticket with a number of the particular Debian package version. The only difference with Docker would be that the developer will send an image signature hash created by Docker. This way there will always be a possibility to authenticate possible and the developer who sent the ticket will be responsible for his image.

The administrator will then pull the image from the dev registry and push it to the staging registry and then to the production registry.

When Kubernetes starts a new pod, the Kubernetes master node downloads the image from registry (if it's not present already) and runs it. This logic makes the registry server a bottleneck for the entire cluster. When the master node cannot pull from the registry server, the application won't start. And even when app is running, if the registry server fails and then Kubernetes decides to migrate this pod from one server to another because of load balancing or something else, application can easily become unavailable. That means that Seznam.cz's Docker registry in the production environment has to run on high availability servers.

The Docker registry is an application which provides an API for clients and the storage itself is delegated to drivers. The default driver is `posix` filesystem. As is said in the manual, this default driver is fine for small deployments and in our conditions will be fine for development environment where physical hard drives will be mirrored. Development registry server does not have to be high availability and there is always a way how to start custom registry even on local machine.

The Docker registry storage drivers provided are [23]:

- **inmemory**: A temporary storage driver using a local inmemory map. This exists solely for reference and testing.
- **filesystem**: A local storage driver configured to use a directory tree in the local filesystem.
- **s3**: A driver storing objects in an Amazon Simple Storage Solution (S3) bucket.
- **azure**: A driver storing objects in Microsoft Azure Blob Storage.

- **rados**: A driver storing objects in a Ceph Object Storage pool.
- **swift**: A driver storing objects in Openstack Swift.
- **oss**: A driver storing objects in Aliyun OSS.
- **gcs**: A driver storing objects in a Google Cloud Storage bucket.

From all these drivers provided the only two options we can use are rados, with Ceph Object Storage and swift with Openstack Swift storage. And because in Seznam.cz there are administrators who are well acquainted with the technology of Swift, we decided that for the production registry we will use swift as a storage driver.

So the final model is shown in the following figure 8.1.

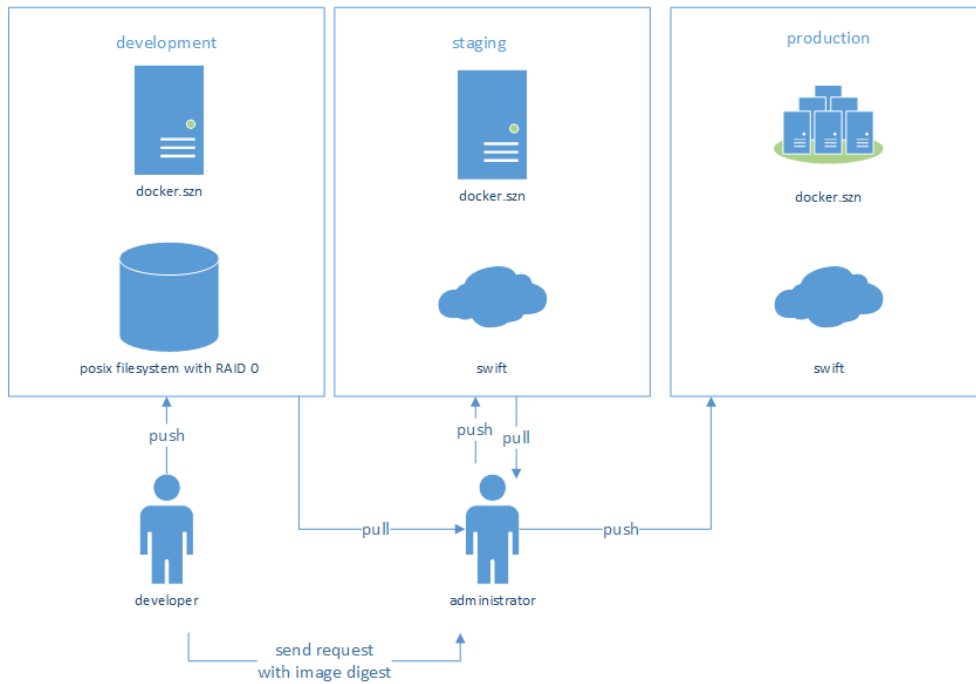


Figure 8.1: Seznam.cz Docker registry architecture



---

## Secrets distribution

Kubernetes has a mechanism for storing secrets — passwords, keys, certificates, etc. Unsurprisingly, they call it the secret.

Objects of the type secret are intended to hold sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a secret is safer and more flexible than putting it verbatim in a pod definition or in a Docker image. [24]

Using secrets is safer than putting those sensitive information somewhere else, like directly to Docker images, to the file system mounted as a volume, to environment variables, etc. The main idea of the secret is to keep them centralized somewhere safe and distribute them only to those master nodes which need them. And also of course to have them on the master node only as long as necessary. For those purposes Kubernetes are using etcd as a persistent centralized storage that ensures high availability through peer-to-peer synchronization between machines. Each master node has a Kubelet daemon which can ask secrets API server for a secret. Communication from the master node to the etcd is encrypted. The Etcd sends the secret object to the master as a base64 encoded string. The size of the secret is limited to 1 MB and the master node will save the secret value to its memory, Not to the file system, which means that when the master node crashes and its memory is deleted, no secret can be compromised. The Kubelet then mounts the secret value to containers in the pod which requested it as a tmpfs filesystem. Secrets in the etcd are divided into namespaces and only pods from the same namespace can ask for them. It is the main responsibility of administrators to check the pod's namespace when they accept it and deploy it to a running cluster.

After consulting the Kubernetes secret with our administrators we agreed that it would be nice to have the same feature for the current virtualization technologies (LXC and OpenVZ containers). The logic can be almost the same as Kubernetes have. And because the Kubelet is a standalone binary which can be called even without Kubernetes and the etcd is also an inde-

pendent component, we decided to build a secrets distribution system for the current solutions at Seznam.cz. Each LXC container or OpenVZ virtual machine will have the Kubelet binary in its image. After installing a new virtual machine, the administrator will add a certificate signed by his team's certification authority, which will be configured at etcd so it can request only secrets belonging to his team. There can be special one way certificates for highly sensitive data as well. After the virtual machine starts, it will start monitoring a known file, such as `/www/secrets/request.json`, where applications or even the administrator of the virtual machine have to specify which secrets are requested. The Kubelet then asks the etcd for them and saves them via tmpfs to a known place such as `/www/secrets/data/`. New applications have to be edited for those new features.

The main advantages of this approach is that all secrets are saved at one safe place with a high availability for the whole company infrastructure and both master and virtual machines do not keep secrets for ever but only on a temporary file system in their memory.



---

# Logging

Logging is quite a big problem in almost every system. At the moment in Seznam.cz we have a special library called `dbglog` which is used for logging from C/C++ and Python. This library is also open sourced [63]. You can configure it to log to `stderr` or to a file. It simply formats the message with data arguments add a general info, like file from which the log message has been called, the line number, the function name, the log level, current time and so on.

The text files take up a lot of space, so we had to think about storing information in a binary form or even compressed. Another idea was to store logs not as simple strings but in a structured form. That would be very helpful for a basic analysis of these debug logs, for example to check the number of records processed etc. We started to develop a complex logging solution quite a while ago. I implemented a library for storing data to binary files. The library is in Go and uses the Apache Kafka [53] file format. Picking the Kafka file format was quite an easy decision, as we wanted to use the Kafka cluster as a logging service where all our applications will send their logs. Then we can decide which logs go to the Elasticsearch [39] for a human analysis and which will go to the HDFS [40] for permanent storage and time consuming calculations. And of course in the future more approaches can come. Kafka file format also supports compression and storing partition key in each message.

A very common problem with logging into file system is rotating files. There are many approaches for this, but `logrotate` or another similar daemon running in the background is not the way we want to go in Docker. The Docker principle is to have only one process in one container. Sure you can run `logrotate` in another container in pod and send signals from one container to the other, but that's generally not a good idea. That's why the library for storing logs into file system also supports file rotation.

So at the moment I have solved the problem with file rotation and storing files from our applications. But how to deal with logs from any third part application? Actually the answer is very easy as well. Does the application

support logging to stderr? If so, just make use of it and through a pipeline to send the stream to stdin for another application which will simply save each line into my library. If stderr is not an option (for example nginx), does the application support syslog interface? If so, do exactly the same, only that the application which reads it will have to expose the syslog interface.

So now I have the files stored at the file system. How to send them to Kafka for further processing? This simple task proved to be quite tricky. There are several daemons which read files and send them somewhere. Kubernetes also have one of them.

## 10.1 FluentD

Fluentd is an open source data collector for a unified logging layer. The Fluentd allows to unify data collection and consumption for a better use and understanding of data [25].

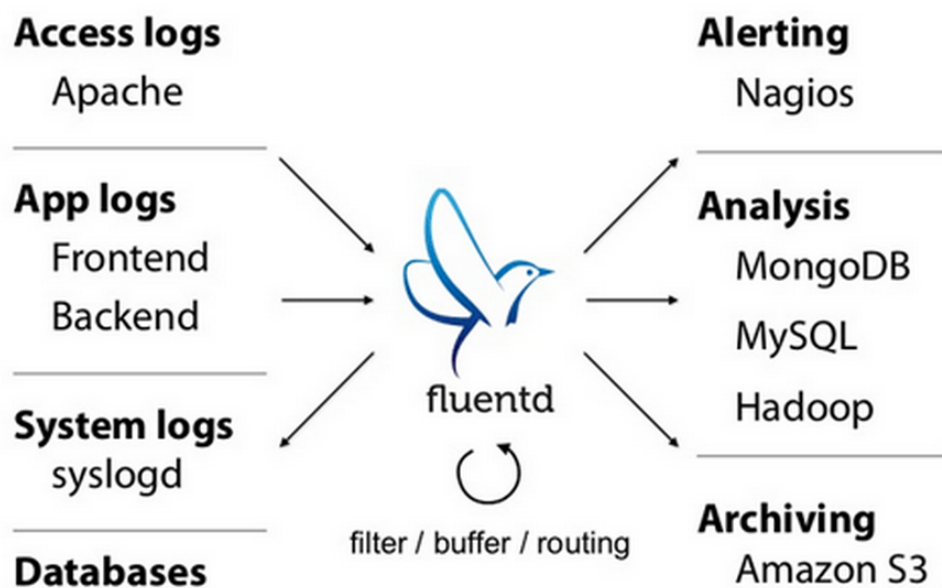


Figure 10.1: Fluentd [25]

The Fluentd is natively implemented in Kubernetes but there is a couple of problems. The first one is that plugins for The Fluentd are written in Ruby [54], which is not used in Seznam.cz, so writing a custom plugin could become an issue. And we already know we would need at least one custom input plugin for reading the Kafka file format. The second problem is that in the default configuration the Fluentd reads from the input file, stores those data in a memory buffer and tries to send them to Kafka. When the Fluentd, its

pod or even the whole node fails, the data are lost and when it starts again it does not know about the power cut and it starts from the last known position in the file. This all means that the Fluentd is not an option for us.

## 10.2 Logstash

Logstash is a tool for managing events and logs. It can be used to collect logs, parse them, and store them for later use (e.g. for searching) [26].

Logstash plugins are written in JRuby so it's almost the same as with Fluentd. We tried to use Logstash in the current conditions at Seznam.cz but it completely failed because of memory demands. Logstash is written in Ruby but runs under JVM, so its memory demands are huge. We also wanted to have one log forwarder from the file system to the Kafka for each pod so that the failure of one pod won't affect other pods and with Logstash's memory usage this is not possible.

## 10.3 Heka

Heka [27] is written in Go, its plugins are written in Go and that is an advantage over Logstash or Fluentd. Heka also supports sandboxed Lua for filter scripting without the need to recompile Heka. There are plugins for Apache Kafka, Elasticsearch and many other outputs. That's why I started to examine Heka more closely.

Heka is a heavily plugin based system. There are six different types of Heka plugins [27]:

**Inputs** Input plugins acquire data from the outside world and inject it into the Heka pipeline. They can do this by reading files from a file system, actively making network connections to acquire data from remote servers, listening on a network socket for external actors to push data in, launching processes on the local system to gather arbitrary data, or any other mechanism.

**Splitters** Splitter plugins receive the data that is being acquired by an input plugin and slice it up into individual records.

**Decoders** Decoder plugins convert data that comes in through the Input plugins to Heka's internal Message data structure. Typically decoders are responsible for any parsing, deserializing, or extracting of structure from unstructured data that needs to happen.

**Filters** Filter plugins are Heka's processing engines. They are configured to receive messages matching certain specific characteristics (using Heka's Message Matcher Syntax) and are able to perform arbitrary monitoring,

aggregation, and/or processing of the data. Filters are also able to generate new messages that can be reinjected into the Heka pipeline, such as summary messages containing aggregate data, notification messages in cases where suspicious anomalies are detected, or circular buffer data messages that will show up as real time graphs in Heka's dashboard.

**Encoders** Encoder plugins are the inverse of Decoders. They generate arbitrary byte streams using data extracted from Heka Message structs. Encoders are embedded within Output plugins; Encoders handle the serialization, Outputs handle the details of interacting with the outside world.

**Outputs** Output plugins send data that has been serialized by an Encoder to some external destination. They handle all of the details of interacting with the network, filesystem, or any other outside resource. They are, like Filters, configured using Heka's Message Matcher Syntax so they will only receive and deliver messages matching certain characteristics.

I developed a custom splitter plugin and a custom decoder in Go, which are able to split and decode data from the Kafka file format. So the default Heka plugin – Logstream Input – reads bytes from the Kafka files, the Kafkalog splitter and the Kafkalog decoder create Heka messages which are send through the filters to a simple encoder and to the Kafka output.

When Kafka ran, everything seemed fine and worked as expected. The problems started when I stressed Kafka. I shutdown some of the brokers and watched how Heka can handle it. Heka noticed that one or more Kafka brokers were down and the rest was in the middle of the leader election, but it didn't stop trying to send messages. It slowed down, because of the timeouts that occurred, but after a while the message that cannot be delivered to Kafka is dropped down and the Input plugin reads new bytes from the file. I tried to simulate Heka failure, simply by Linux kill command.

The Logstream input plugin keeps a journal file. In the journal there is the file name, the offset of the last read message and the control hash. But the problem is, that input plugins do not cooperate with outputs. Heka is highly expandable with plugins. One input can generate message to Heka's pipeline and more outputs can read them and do something with them. Filters can drop messages or create new messages so there is no simple way to synchronize the input and the output plugin. Which means there is a risk of losing messages.

I did a research about Heka's reliability and found a couple of some e-mails and discussions where the Heka authors are saying that 100 % reliability was never their goal. But we need it at Seznam.cz as we can't say that some logs will possibly be lost. So I try to fix this.

The Kafka output plugin uses the Sarama library from Shopify [28] to communicate with Kafka. There is a synchronous and an asynchronous Kafka pro-

ducer. In the output plugin the asynchronous one is used. The asynchronous producer is faster, because it does not wait for errors or success confirmation. All messages are handled in separate goroutines<sup>1</sup>. Producer errors are handled in output plugin, but only to be logged and dropped down. I fixed this with a special error channel addition. When an error occurs, it is sent to the error channel and the main goroutine which process Heka messages starts fixing it. When there is an error I don't want to process any other messages from the Heka pipeline, the backpressure is desirable. I create one extra synchronous Kafka producer with exactly the same configuration as the main (async) one and the error is sent through it. The maximum number of attempts is configurable and can even be set to infinity. Dealing with the error through the synchronous producer will not read from the main message channel, which can fill up eventually. This causes a backpressure and Heka will stop in such case (of course, only under the condition that no other output processes the input messages).

I have tested this by writing a Kafka consumer in Python and uploading files with a few millions of numbered messages. The consumer reads everything from the special topic and looks for holes and duplicates in sorted sequence of messages. I randomly shut down Kafka brokers, even whole Kafka, simulating packet lost with Linux iptables DROP directives and watched consumer's statistics. It turned out that with a few millions of messages there is a few thousands of duplicates but no miss. Duplicates can also be caused by consumer, because Kafka's philosophy is to deliver each message at least once. And duplicates are no problem for us, because we can easily discover them. This fix seems quite useful and generic, so I will try to send it to the upstream as a merge request.

Fixing this issue will only try to repeat errors when they occurs. This does not solve the next problem, when Heka input plugin reads something from a file, it confirms the new offset to its journal file and sends the message to the Heka pipeline. If Heka fails now, the memory buffer will be lost and after a new start the input file will seek to the position from the journal file. I need to figure out how to fix this too.

The problem is, that one input can be processed by more outputs, and also more inputs can be processed by a single output. The input and the output plugins don't know about each other. I think there is not one generic solution for this problem. The best what I could come out with is the following idea:

- Each input plugin will be working as is plus it will be adding filename, offset and its name as fields to the message.
- The output plugin will be also working as is plus it will read the success message from Kafka, confirming that message was delivered to it. Output saves those metadata sent from input to its private variable and

---

<sup>1</sup>A goroutine is a lightweight thread managed by the Go runtime [62].

once in a while it sends a special message: an acknowledgement consisting from the filename, the offset and the name of the input plugin.

- I will write a new plugin, the Checkpointer, which will consume acknowledgements from the output and save checkpoint files per each input. This plugin has to know which outputs consume messages from which inputs to be able to successfully maintain acknowledgements from more output that consumes from single input.

This theoretical idea might be good, but it is unfeasible in practice. The main problem is, that output plugins are not allowed to generate new messages. Also the condition to know about all other plugins is senseless in Heka.

So I have to update my idea to be implementable in Heka and useful for Seznam.cz. Our applications logs through my library, which rotates file and is using our naming convention: `datetime-timezone-rotate_interval-component_name-postfix.szn`. One application can generate more files with different postfix value. For example `szn-fulltext-NAME-dbg` and `szn-fulltext-NAME-event`. In the log directory, there is a `kafkafeeder.yaml` file – with our generic configuration where to send files, how long to retain them and so on. That means in our conditions it is possible to always have one input for one output. The problem that output plugins are not able to send new messages can be solved by implementing the checkpointer logic right into the Kafka output, so it will be saving checkpoint files to the file system.

There have to be written a new binary, which will start, copy checkpoints to journals, transform Kafkafeeder configuration to a Heka specific configuration format, start Heka with some predefined options, watch checkpoints and data in a separate goroutine and delete old (and successfully uploaded) log files from filesystem.

This solution will be 100 % reliable but it is also Seznam.cz specific, because our custom binary (let's call it Kafkafeeder) will transform our custom configuration to the Heka format and it will create one Input plugin for one Output plugin with a unique name, so checkpoints can be saved and reused as journal files of those Inputs.

### 10.3.1 Kafkafeeder

I started designing this application in Go. The Kafkafeeder has to:

- Copy checkpoints to journal directory
- Start Heka daemon (`hekad`)
- Monitor log directory for changes
  - Reload when it finds a new `kafkafeeder.yaml` or when an existing one disappears

- Convert the `kafkafeeder.yaml` custom configuration to the Heka format and reload hekad
- Monitor checkpoints and delete old and successfully uploaded logs from the directory

The application is made as a five goroutine architecture, where the main goroutine initializes all the necessary objects and starts all other goroutines.

The cleaner goroutine is responsible for removing old logs, which have been successfully sent to Kafka. Successful dispatch may be tracked from checkpoint files generated by heka. All files with their last modification time older than the retention duration in the configuration file will be deleted.

The signal goroutine is waiting for system signals and handles them properly. When `SIGTERM` is captured, the main shutdown function is called and the whole application stops safely. When `SIGCHLD` signal is captured forked child is dead and so I have to check if it should be running. If so, a problem occurred and the main shutdown function is called.

The watcher goroutine periodically watches the directory with logs. When `kafkafeeder.yaml` file is found it is registered to LogManager object. After all new logs are registered, the `keepValid` function is called so the LogManager will keep only those records that actually exists. If something was changed, the reload is called.

The hekad goroutine is responsible for copying checkpoint files to the Heka journal files and starting, stopping and checking `hekadCmd`. Also when the reload is called, the hekad goroutine converts all logs from LogManger to the Heka specific configuration.

`HekadCmd` is a wrapper around the actual hekad process. This wrapper sends signals to forked process, captures its standard output and error and logs all messages via the internal logger.

## 10. LOGGING

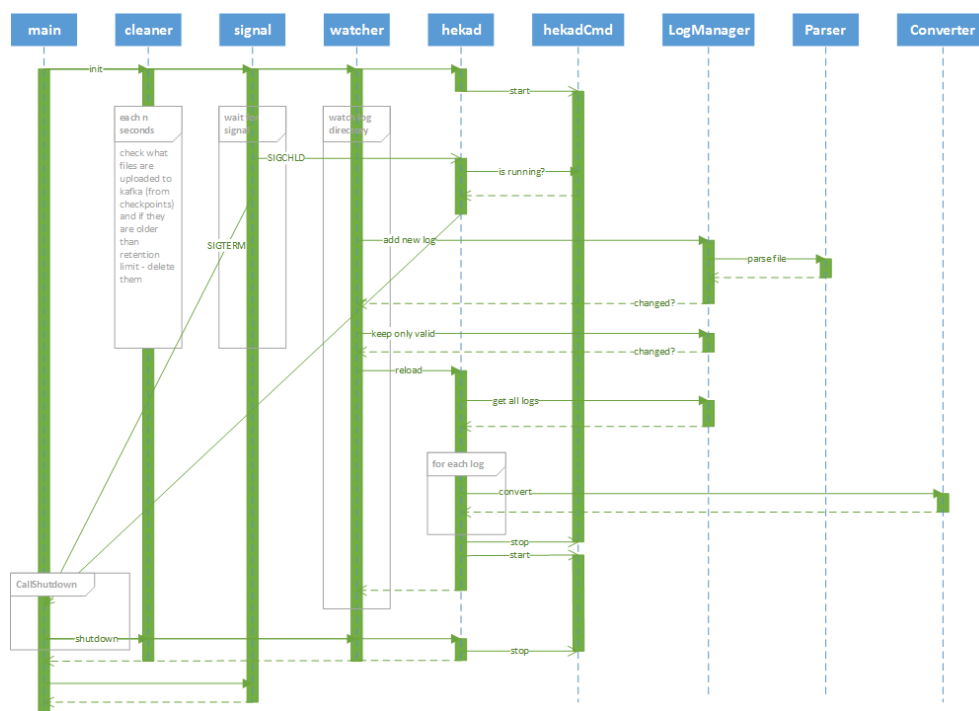


Figure 10.2: Kafkafeeder architecture



---

# Security

As was written before, at the moment masters and virtuals in production environment are under the administration of administrators. When a security issue is discovered, they can fix it in a matter of minutes. But on the other hand developers are forced to pack each app into a Debian package. And there are applications from many different corners, Python uses pip [56] for its packages, nodeJs [55] has npm [67] repositories, Go is built from source code and so on.

With the advent of Kubernetes it will no longer be sustainable to check if every piece of code in production is packed in a Debian package and uploaded to custom repository, where it can be easily tracked if a newer version with security fixes is released.

Because of that there has to be created a new mechanism how to check:

1. What is installed and will be running in a Docker image (system, applications, dynamic libraries, ...)
2. Using what and where was it built (gcc version, Go source codes, static libraries, ...)
3. Where is the Docker container running at the moment

Those easily looking conditions are quite hard to satisfy in practice. Seznam.cz at the moment is using custom system called Puzzle where everyone can find on which virtuals is his component running, what packages and their versions are installed on specified virtuals and on what virtuals is installed specified package and version. Those information are delivered to system backend via script called self-checked that have to be installed on each master and is monitoring its virtuals and software installed in it.

After consultation with our security admins we decided that same function is needed also with Docker cluster.

The suggested solution is to create a mechanism that will sniff Docker image during pulling from development registry and pushing it to staging registry

and divulge all necessary information about its content. Those information together with image digest will be sent to the Puzzle system API for future tracking. There have to be tracked not only version of debian packages but also pip and other python modules versions, npm packages, Bower [64] packages... All other packages apart from ours have to be tracked as well. So we will have to build cache server for each of those repositories and run it. When developer wants to use some library (in production, developing won't be limited) he has to add it to this cache under his name and from now he is responsible for it, that means when newer version will be found or some security issue will be discovered administrators have a privilege to demand update by this developer and rebuild all dependent images (which will be tracked by the Puzzle system).

Built information are harder to get. A common scenario is to have separate image or virtual for building. When virtual is used, there is often one virtual for building more images that means simple imprint of system state will be unnecessarily comprehensive. Also when using Go, build is made from source code so there is no information about version of code.

For start imprint of this development environment have to be sufficient. For Go we can standardize using glide [65] for vendor dependencies tracking so each project will have `glide.yaml` file with dependencies and after built there will be created `glide.lock` file with exact commit digest of vendor version control system. For this information to be responsible developer who will have to put it all in a simple text file to its production image on well-known place such as `/www/built-info.json` for example. Those files then will be read and send to Puzzle system as well.

Getting runtime information from orchestration system like Kubernetes appears to be the easiest of all conditions. Kubernetes has an API server where it is possible to get information about running containers.

With all those information in a custom system we can easily track if there is a new security issue and what version fixing it and if it is backward compatible. Debian offers such mechanism and Seznam.cz is using it right now, for pip and other management systems we have to find such mechanism and implement it to our environment. This task will be a responsibility of our security administrators.

A scenario when a security issue in a system package occurs (such as Heartbleed) is: find all images that uses this affected package, run script that will create new Docker image from base developer image and run installation of fixed package version. An example template of Dockerfile may look like this:

```
1 FROM developer_image
2 RUN apt-get update && apt-get install -y openssl=1.0.2-fix
```

Listing 11.1: Dockerfile snippet

---

This process will be automatic of course.

However, this solution is only temporary, the future plans are to create an independent building environment under control of administrators (or integrators perhaps) where automatic builds of images will be run periodically and precisely because of information provided from developers, exact built environment can be generated for each image that guarantees smooth built process.

On a meeting with our system administrators one more suggestion have been put on table. Seznam.cz have to build its own base image that will contain new packages with all system fixes. This base image will be generated automatically each hour and pushed to all Docker registries. Developers will have to use this base image or scratch image when no system is required (running some c++ binaries for example). Using any other system than that one from base image (Debian) won't be allowed.



---

## Monitoring

For monitoring application metrics such as count of requests, returned statuses, average request duration etc. we will use Prometheus. Prometheus is an open source systems monitoring and alerting toolkit [29]. Implementing Prometheus handler to webserver is really easy because there are client libraries for Python and Go. Those metrics are then easily scraped and stored in centralized database where some other alert system may be configured.

Prometheus has also prepared a Docker image so it can be run in Kubernetes cluster.

System logs will be collected with Heka. Heka provides various input plugins like syslog, statsd or simple tcp input, while Kafka can be used as an output just like with all our others logs. This way we will have all types of logs centralized in the Kafka cluster and we can decide what to do with them next (and even change that decision later).

Metrics and system logs may not necessarily be stored forever. They have their usage in the real-time and so they probably will not be uploaded to HDFS for some future analysis. System logs will be send to the Elasticsearch and they will be visualized in Kibana [57]. We already have the Elasticsearch cluster in Seznam.cz and the Kibana running so it is no problem to add those logs here.

Metrics will be stored in Prometheus and Grafana [58] will be used for data visualization. Prometheus also offers alerting system so each application can have its separate rules and when something goes wrong an alert may be issued.



---

## Static webpages content

Deploying static resources the same way as before is no longer possible. We have to solve this problem with a different approach. There are at least 2 acceptable solutions. The first one is to upload static files to our CDN. Seznam.cz already has a custom CDN which is used for delivering images and other resources. But at the moment we are not using the CDN for delivering CSS and JavaScript files because there was no such need. Using this solution would mean to discard current debian packages for static resources and start uploading those files to the CDN instead.

The second possible solution is to deploy static resources in a separate pod with their own nginx server which will be serving them. This solution seems as a little overkill to me because when a new version of CSS files will be created we will have to deploy new pods, wait until all running pods with static files are replaced and after that start deploying pods with new backend. The backend pods will also contain an nginx proxy because of SSL termination and security issues, so it seems unnecessary to run another nginx instance solely for the purpose of serving static files.





---

## Testing application

Before we can run the Kubernetes cluster at Seznam.cz and start updating all applications for it, it has to be tested so we can say it will suit our conditions. For that testing I developed an application that'll push it to its boundaries.

The hardest task of a programmer is naming things [30] but in this case my team leader came with an idea – the Tarsier. Tarsiers are small primates with enormous eyes. Each eyeball is large as its entire brain [31]. That means they will probably not invent anything new and useful, but they can watch and see everything – which quite fits for my testing application needs. It won't be doing anything extraordinary, it just has to test the cluster and find its weaknesses.

The basic requirements on the Tarsier are to heavy load the machine resources, to simulate high CPU and RAM usage, to simulate I/O operations (so it will have to request permissions on some persistent storage), to raise controlled faults, deadlocks and segmentation faults and finally to simulate a network communication.

Of course the Tarsier will have to log its activity and probably also expose its inner statistics and metrics.

When designing the Tarsier I have to think about implementing new functions in the future, so it should be modular. User need to be able to control the Tarsier's behavior from the outside, because deploying a new image just because of a different configuration to make use of another set of its skills would be slow and inefficient.

Another important thing is that one Tarsier will hardly be able to push the cluster to its boundaries, but if we employ many monkeys at once that can communicate with each other and start using the network at the same time for example, that is something that should be able to easily find the limits of the Kubernetes cluster.

## 14.1 Designing Tarsier

I designed the Tarsier as a modular application. However, as there are no dynamic libraries in Go that could be joined together during the runtime, every application with all its dependencies is built from the source. So I created a plugin interface and a plugin registration function that will store plugin's factory method, so Tarsier can invoke the plugin later with the desired configuration.

As the Tarsier will have to do many different tasks, I decided to separate each task to the Command interface and plugin that has some commands will register them.

```
1  type Plugin interface {
2      Init(config interface{})
3  }
4  type HasConfigStruct interface {
5      ConfigStruct() interface{}
6  }
7  type HasCommands interface {
8      Commands() []Command
9  }
10
11 type Command interface {
12     Execute(data interface{})
13     Name() string
14     Description() string
15 }
16 type HasDataStruct interface {
17     DataStruct() interface{}
18 }
```

Listing 14.1: Tarsier plugin's interfaces

For remote controlling the Tarsier I use HTTP requests, so the Tarsier is a modular API server, where plugins can register themselves and their commands and the Tarsier then waits for orders and executes the appropriate command when invoked. Each plugin can claim self-specific configuration and each command may demand specific data. There can also be plugins with no commands because it is possible that in the future the requirements on the Tarsier might change.

At the moment Tarsier starts as a HTTP server with one handler on the `/exec` URL and it accepts YAML POST requests (and because valid JSON data comply with the YAML specification, it accepts JSON POST requests as well). The structure of the body is the following:

```

1 command: "heavy_load/spin_cpu"
2 data:
3     duration: "15s"
4 wave:
5     remain: 3
6     buddies: 10

```

Listing 14.2: The `heavy_load/spin_cpu` command body structure

If the command named `heavy_load/spin_cpu` is registered and if it implements the `HasDataStruct` interface, the data from the body will be unpacked to the `DataStruct` method returned value and the command will be executed with that structure. This will happen in a separate goroutine so the handler can also process the wave part. The wave remain is a number of how many waves are remaining to do. If it is greater than zero that means this Tarsier will have to send this request beyond and notify 10 its buddies with the same data as he gets. Obviously the wave remain value will decrease.

The waves are spread exponentially and it is possible that some request will come back to the same Tarsier that sent them but that does not matter because I want to test the cluster and some random behavior is even welcome. Let's see another example command:

```

1 command: "persistent_storage/write"
2 data:
3     amount: "100 MB"
4     files: 10
5 wave:
6     remains: 3
7     buddies: 3

```

Listing 14.3: The `persistent_storage/write` command body structure

This will start writing 100 MB into 10 files in parallel on each Tarsier. And also this command will be sent to 3 more Tarsiers randomly chosen from friends list. The flow of the wave with 9 tarsiers might look like in the following example 14.1.

## 14.2 Tarsiers synchronization

To be able to discover all the Tarsiers and to help each one of them find the others so it can send waves to them I will use Consul [32]. Consul is designed for service discovery and makes it easy to register services and to discover them later. Each Tarsier will register itself at startup and then it will periodically check its buddies. To ensure that Consul will provide only active

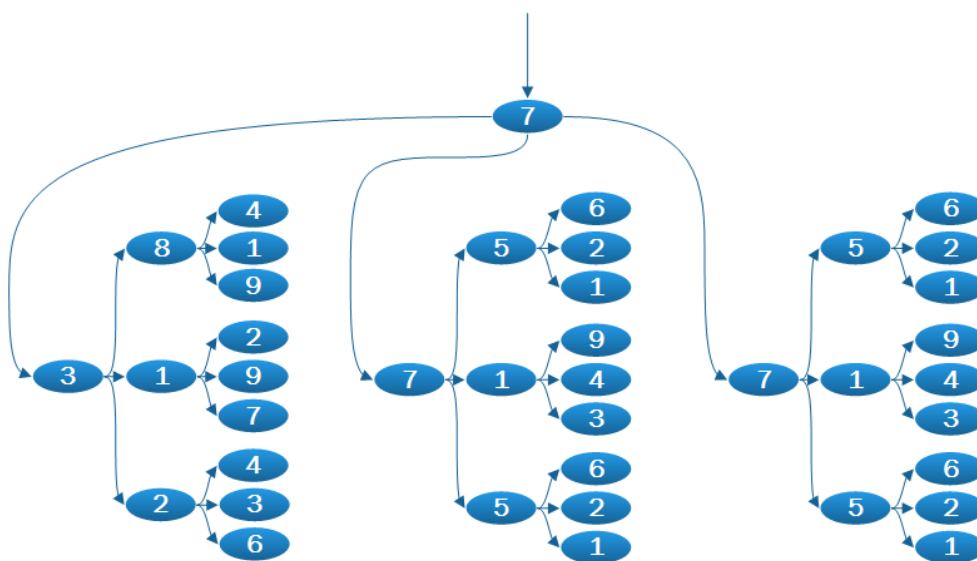


Figure 14.1: Wave propagation in Tarsier

Tarsier instances, it has multiple types of health checks. Tarsier will have to provide some service stats.

At first I wanted to just add some special URLs for this service stuff such as `/health_check` and `/metrics` but then I realized that in the future those URLs may be needed and also those URLs don't have to be accessible from the public network so I designed Tarsiers with another extra interface on its own port where private data will be provided. There will be health check and metrics handler with Prometheus statistics about how many requests have been served and what commands were executed and if they were successful. I also created a command handler where all the registered commands with their names and descriptions are provided. Because there will be plugin to simulate heavy load of memory and CPU I add a handler that shows runtime statistics from Go. There can be seen how much memory is allocated to the Go runtime, a garbage collector statistics and much more interesting data.

## 14.3 Tarsier plugins and their commands

### 14.3.1 Heavy load

Plugin for simulating heavy load of memory and CPU

**Gobble RAM** will gobble as much RAM as is specified in the data. When this amount of memory is not possible to allocate it will try to resize the amount with the specified ratio and return how many bytes it finally allocated. Memory is allocated via `mmap` system call so the garbage

collector will not free it and is filled with zeros otherwise system will not physically allocate it.

**RAM stats** will show allocated mmap regions with their sizes.

**Free RAM** This command will unmap allocated region. Its data is the index of the region obtained from the RAM stats command or -1 which means to free all allocated spaces.

**Spin CPU** spins the CPU for the specified time. The spinning is simply done by an infinite loop with a computation (multiply and division) with float64 numbers.

### 14.3.2 Faults

**Deadlock** will create the specified number of goroutines that will be set to be sticky with system threads (otherwise Go runtime is allowed to move them all into single system thread, so it has no effect) and deadlock them all waiting for mutex unlock. After the defined time the handler will unlock the mutex so all goroutines will finally end.

**Segfault** will simulate segmentation fault. It uses Go unsafe package pointer to variable, which is moved out of bounds and dereferenced. The whole Tarsier will fault and it is expected that Kubernetes will start new instance.

### 14.3.3 Net

This package is here to wrap all the network commands. There are no commands for writing or reading from the network but that can be easily simulated with a dummy command to the Tarsier where waves count will be high enough.

**Dial** will open a new network connection to the specified address. It can open a new TCP or UDP connection and also this connection can be established directly to an IP address or to a domain name with its resolving. Those connections are set not to timeout and are not closed.

**Stats** will show the network connections usage.

**Close** will close the specified network connection from the Stats command. When -1 is set it closes all connections.

### 14.3.4 Persistent storage

Plugin that is working with persistent storages, typically hard disks. This plugin needs to get directory from configuration of Tarsier otherwise system temporary directory is used.

**Open FD** will open as many file descriptors as defined.

**Read** This command will read the specified amount of bytes from defined file. Typically `/dev/urandom` is set as a file. There is also an option how many concurrent readers are supposed to be created.

**Write** will write the specified amount of bytes to opened file descriptors. There can be set how many concurrent writers are supposed to be created and each one is writing to its own file. If there is not enough file descriptors ready, new ones will be opened.

**FS stats** will show usage of file descriptors, their names and sizes.

**Close FD** will close the specified file descriptors from the FS stats command. When `-1` is set then all file descriptors are closed and the files are deleted from the hard drive.

### 14.3.5 Sleeping beauty

Plugin for simulating requests delay.

**Sleep** will answer after the specified time expires.

---

## Testing the cluster

Running applications in Kubernetes means to create some specifications for them. For testing the cluster with my Tarsier application I will need to have the Consul running at first.

I have prepared the replication controller configuration for Consul and also the service configuration for it, so it will have the same IP address regardless of which machine it is running on. The Consul will be used for a service discovery among Tarsiers. There is no need to use persistent storage for Consul, holding instances in memory is sufficient enough.

Before Tarsier pod can be created I have to register secrets for it. There will be two of them. The first secret is a SSL certificate so our service port can be run on HTTPS. The second secret is the Tarsier configuration. It is a good idea to use secrets for configuration because there can be different secrets with different configurations on the development and the production cluster and also we don't need to rebuild the whole images just because one text file changed.

With the secrets set up, it remains to solve the persistent storage. For logs I will use the `hostPath` volume which is a directory on the physical machine. There will be uniform policy that perhaps in `/www/logs/` all pods will store their logs. This way it is provided that logging from the application is fast (to the local drive) and logs will remain there even when the pod dies. Also each pod should use the Kafkafeeder to feed logs to Kafka and delete them after successful upload.

For persistent storage plugin of Tarsier I have to use a real persistent volume. There are many applications in Seznam.cz which need a large amount of data for their start or/and which produce a large amount of data. For those data we have to choose a persistent storage where pod traveling from one machine to another will not be a problem.

I started 2 GlusterFS [61] servers and created endpoints and service configuration for them. Now I can set a volume claim in the Tarsier replication controller configuration and use it without knowing the exact location of the

GlusterFS and other paths.

In the Tarsier RC configuration will be more than one container. The second one will be the Kafkafeeder, which also produces logs, so it stores them to `/www/logs/kafkafeeder` as well. Then it loads this directory and starts watching it and uploading logs to Kafka. After setting those volumes I realized that the current usage, where application creates symlink from its log directory to the `kafkafeeder.yaml` file in the configuration directory, is no more useful. The `kafkafeeder.yaml` file which tells the Kafkafeeder where and which logs to upload is stored as another secret in Kubernetes and mounting this file as a volume with logs from the `hostPath` is not possible. I solved this problem by adding another container to the pod, which is created from a simple image with a single static application. User passes the source and the target file to that application and it simply copies it. The snippet of the replication controller is as follows in the listing 15.1.

With those configurations prepared, the last one missing is the Tarsier service. Then I will scale the Tarsier to many instances and start sending commands to its interface.



---

```

1 containers:
2   - name: tarsier
3     volumeMounts:
4       - name: tarsier-logs
5         mountPath: /www/tarsier/logs
6
7   - name: kafkafeeder
8     volumeMounts:
9       - name: kf-logs
10        mountPath: /www/kafkafeeder-beta/logs/
11
12  - name: cp
13    args:
14      - "/src/kafkafeeder.yaml"
15      - "/dst/kafkafeeder.yaml"
16    volumeMounts:
17      - name: tarsier-kafkafeeder
18        mountPath: /src/
19      - name: tarsier-logs
20        mountPath: /dst/
21
22 volumes:
23   - name: tarsier-logs
24     hostPath: { path: /www/logs/tarsier/ }
25   - name: tarsier-kafkafeeder
26     secret: { secretName: tarsier-kafkafeeder }
27   - name: kf-logs
28     hostPath: { path: /www/logs/ }

```

Listing 15.1: Snippet of replication controller configuration



---

## Benchmarking Python and Go application servers

Traffic on Seznam.cz applications is high and it moves around a few thousands requests per second per one instance of an application. Python [34] is used for most of our application servers, especially the Tornado framework [50] which can handle requests asynchronously and is more lightweight [52] than frameworks like Django [48] or Flask [49]. I decided to benchmark such Python server and compare its performance with a server written in Go [33]. Go is a relatively young programming language, yet there are many big applications that use it. Kubernetes and Docker are both written in Go. There are also many libraries for Go, which makes it more interesting for us than Rust for example.

I developed two simple servers which handle GET requests, generate HTML output based on a template and return the response. In Python I used the Jinja2 [51] templating system, in Go I relied on the built-in `html/template` package. Both servers compose the output using a layout template and both render one variable in the template.

I started those servers in Kubernetes and exposed their ports as a service. Then I had to choose which benchmarking tool to use, so I picked wrk [35].

Wrk is a modern HTTP benchmarking tool capable of generating a significant load when run on a single multi-core CPU. It combines a multithreaded design with scalable event notification systems such as `epoll` and `kqueue` [35].

Wrk allows to modify the number of client threads and the count connections opened simultaneously. I started with 50 threads and 2000 connections opened at once. Each test was firstly start with 10 second interval for warming up the server and then repeated 3 times with 2 minute length.

I started both servers in Kubernetes and allowed them to use only one CPU core. Then I was adding cores to the servers and ran this test again and again.

For the testing I used a bare-metal server with the Debian Jessie operating

system installed. The hardware configuration of this machine was 24 Intel Xeon processors at 2.27 GHz and 32 GB of RAM. Both servers were running in Docker on one machine and I used another one to run the benchmarks one after another.

The request rate per second is shown in the graph 16.1 and the average latency in the graph 16.2.

From the graphs we can see that the Go application server is far much faster and is scaling linearly with the CPU count. The top boundary of the scaling was about 33 000 requests per second and was most likely held back by the network limits in the developer VLAN where my machines were placed. Python scales well in the beginning but with an increasing CPU count it starts to scale more slowly. Even with all of the 24 CPUs involved it didn't achieved the performance of the Go server and that is a good reason to start writing our application servers in the Go language.

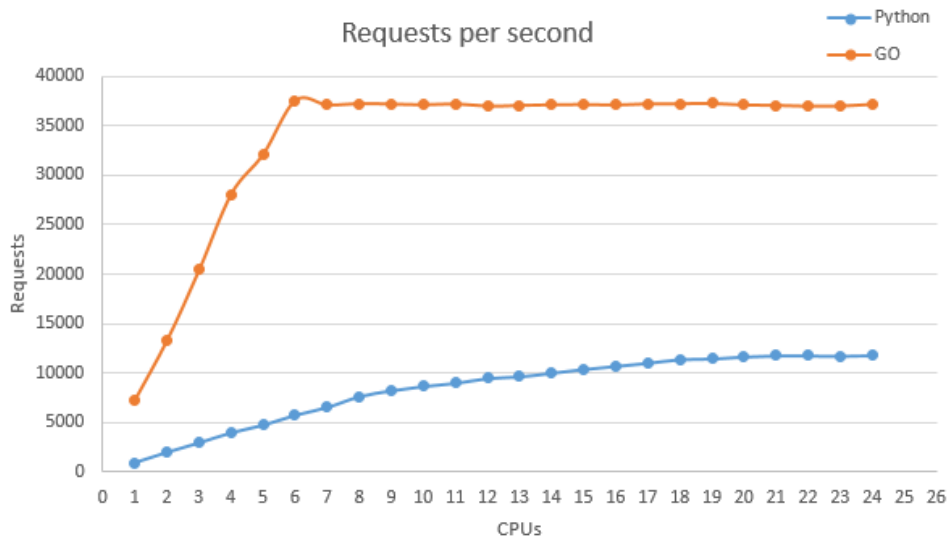


Figure 16.1: Benchmarking Python and Go – Requests per second

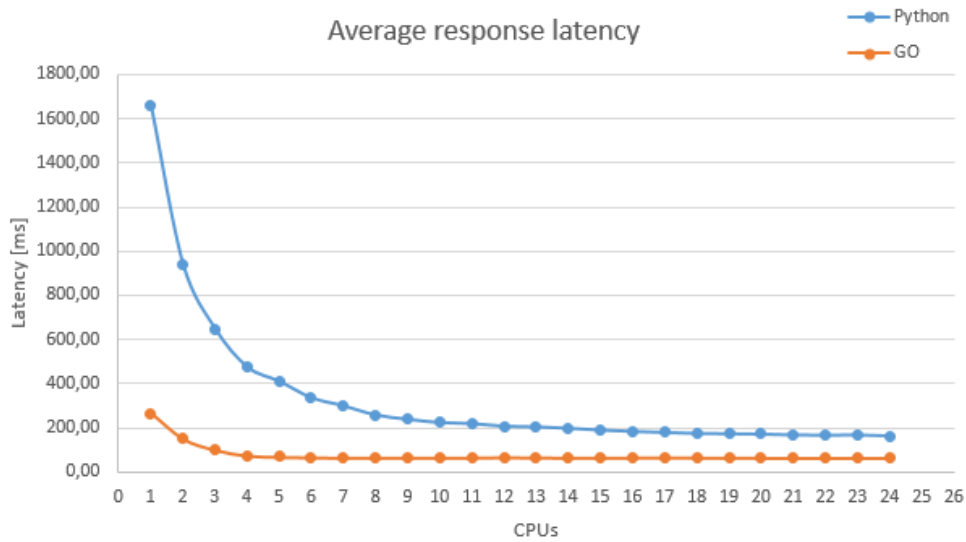


Figure 16.2: Benchmarking Python and Go – Average response latency



Figure 16.3: Benchmarking Python and Go – Transfer per second



---

## Benefits for Seznam.cz

In the chapter “Seznam.cz nowadays” I described how the developers are working today, what do they have to do and how the administrators are managing the machines. What is the motivation to switch to a solution based on Docker and Kubernetes and how to spread this new idea around? Obviously, apart from all the problems we identified before, there also have to be certain key benefits for both the developers and the administrators, otherwise it wouldn’t make sense to use it.

The biggest benefit I see as a developer is that when an environment is created and deployed in the production environment, the developers can still access it easily. That is something I miss a lot nowadays. The administrators prepare the virtual machine in the production using their Salt [68] prescriptions and then install packages from the developers. The developers are building and testing packages on their own virtual machines they create. There can be differences in many things from the network settings to different source list and different versions of libraries. It is also difficult to maintain virtual machines for building and testing for developers who could spend that time coding instead.

Docker solves those problems. Developer knows which packages in which version he needs, so he creates a Dockerfile with it. His image will be built from a base image provided by administrators where all the necessary settings are done and the same image will be used both in the production and in the testing environment.

Docker also solves problems with library versions. Typical example may be one virtual machine where one service is running which consists of many packages and many smaller micro services. Each of this micro service is maintained by a different developer from the same team. When one of them changes somethings and wants to upgrade his micro service but he depends on a newer library, all other micro services will have to be updated as well. But with Docker, he just pushes the new image that will be updated in pod and no one else intervention is needed.

For the administrators the deployment will be easier as well. The developer defines the whole environment in a single image, so they just have to start it. The problem will be with debugging issues that occur in operation, because nowadays they can connect to any server and use debug tools they want and know. With Docker images in Kubernetes they first have to find the right container so they can attach to it, but if that image was created from scratch, no tools are provided there. In the beginning of our Kubernetes experience we should probably create all images from a base Debian image tailored to our needs so the administrators can use the same tools that they are used to. Then in the future we can move to more lightweight solutions.

Another inconvenience that the Kubernetes solution will solve is the secrets and configuration distribution. When the product manager wants a configuration change, for example setting a lower bound of a delivery score, the developer currently has to repack the application and issue a request ticket. The administrator then stops traffic to one server, installs the new version and restarts the application. After confirming that everything is alright, he can move on with the rest of the servers. With Kubernetes the developer will simply update the configuration, generate secret, push it to Git and send a merge request to the administrator. When the administrator accepts it, he updates the secret in the Kubernetes cluster and starts rolling update (or just sends signal to container, if the application supports configuration reload when running). Also SSL certificates can be stored in the Kubernetes cluster for development and production.

With Kubernetes the developers don't need to maintain virtual machines for testing anymore. The administrators start two (or even more) instances of the Kubernetes cluster in the production and in the testing environment. The same pods, secrets and other resources will be deployed to those clusters.

Economic benefit is also that fewer servers is needed because they will be used more effectively. We can scale applications fast and more adaptively. Traffic will be monitored and examined and there surely be services that have minimal traffic in night so their number can be reduced dynamically.

The idea for the future not so far away is to use the same cluster which means the same machines in production delivery (services such as homepage and others) and for internal tasks like counting signals of webpages, machine learning of our algorithms and so more. If we group all machines in the data centre to one cluster and define proper policies for pod preference and priority, we can use the whole potential of this cluster at night and still have sufficient availability for users who access our services at the same time. This is the future goal were we are heading for.



---

# Conclusion

The goal of the suitability analysis of Kubernetes for Seznam.cz was to become familiar with the basics such as using Docker and virtualization in general. That's why the beginning of this thesis is dedicated to the theory about Docker containers and using Kubernetes for orchestrating them.

The next step was to employ those information in Seznam.cz's specific environment. I mapped and examined the current situation of application development and deployment at Seznam.cz and was wondering about how to upgrade it and how to make use of container virtualization there. I focused on possible problems such as creating the Docker registry, how to deal with the security and contents of images, how to log and where to store logs for further analysis and I also had to have in mind that containers are able to move between machines but logs have to be processed correctly. I had to propose how to monitor the applications in the containers and also the containers themselves. And I had to find out how to deploy the static content of the webpages using load balancers.

I successfully dealt with all of those issues and after consulting with our security administrators I suggested how to secure and monitor what is in the images that we are running. I suggested the architecture of the Docker registry we have to build. This architecture achieved high availability thanks to the Swift storage on which the Docker registry will be running in each data centre. I explored different network management solutions that are possible to use in Kubernetes and chose to start the cluster with the flannel. The flannel has a significant overhead compared to other possibilities but it is sufficient for the purposes of my thesis.

It is important to know how the new Kubernetes cluster is behaving under heavy load and in other specific situations such as deadlocked applications, too many open files or network connections, so I developed an application for testing such cluster. This application uses secrets with SSL certificates, secrets with configuration and produces logs. It also uses a persistent storage, for which I utilized GlusterFS servers. The application provides Prometheus

## CONCLUSION

---

metrics on its service interface. Moreover I created another application that watches a specified directory and sends logs to kafka for further processing. I had to resolve many problems with reliability and performance and I handled them successfully and implemented the Kafkafeeder application which will be present in each pod for uploading the logs.

In the end I focused on better repeatability of the cluster creation procedure and I wrote scripts that simplify installing it again later or adding new nodes to the existing cluster. I wrote the Kubernetes configurations for my applications, creating services, secrets and other prescriptions used in Kubernetes.

During my work on this thesis I found answers for many generic issues associated with starting the Kubernetes cluster while only a few of them were specific for Seznam.cz, so I decided to open source all my code and images I created. Those applications, images and the Kubernetes configurations can be used as examples for further development and can help others as well.

---

# Bibliography

- [1] CoreOS. *flannel*. [online] [cit. 2016-05-09]. Available from: <https://github.com/coreos/flannel>
- [2] GitLab.org. *GitLab Documentation*. [online] [cit. 2016-05-09]. Available from: [http://doc.gitlab.com/ce/ci/quick\\_start/README.html](http://doc.gitlab.com/ce/ci/quick_start/README.html)
- [3] Google, Inc. *Kubernetes*. [online] [cit. 2016-05-09]. Available from: <http://kubernetes.io/>
- [4] Google, Inc. *Kubernetes – Guides*. [online] [cit. 2016-05-09]. Available from: <http://kubernetes.io/docs/>
- [5] VMware Inc. VMware. [online] [cit. 2016-05-09]. Available from: <http://www.vmware.com/>
- [6] Parallels, Inc. OpenVZ. [online] [cit. 2016-05-09]. Available from: <https://openvz.org/>
- [7] Oracle Corporation. VirtualBox. [online] [cit. 2016-05-09]. Available from: <https://www.virtualbox.org/>
- [8] Canonical Ltd. Linux Containers. [online] [cit. 2016-05-09]. Available from: <https://linuxcontainers.org/>
- [9] CoreOS. rkt – App Container runtime. [online] [cit. 2016-05-09]. Available from: <https://github.com/coreos/rkt>
- [10] Wallner, R. Linux Containers: Parallels, LXC, OpenVZ, Docker and More. [online] [cit. 2016-05-09]. Available from: <http://aucouranton.com/2014/06/13/linux-containers-parallels-lxc-openvz-Docker-and-more/>
- [11] Docker, Inc. *Docker*. [online] [cit. 2016-05-09]. Available from: <https://www.docker.com/>

- [12] Docker, Inc. *Docker Docs*. [online] [cit. 2016-05-09]. Available from: <https://docs.docker.com/>
- [13] Kukrál, T. Kubernetes – úklid mezi kontejnery. [online] [cit. 2016-05-09]. Available from: [https://www.linuxdays.cz/2015/video/Tomas\\_Kukral-Kubernetes\\_uklid\\_mezi\\_kontejnery.pdf](https://www.linuxdays.cz/2015/video/Tomas_Kukral-Kubernetes_uklid_mezi_kontejnery.pdf)
- [14] CodeOS. *etcd*. [online] [cit. 2016-05-09]. Available from: <https://github.com/coreos/etcd>
- [15] Google, Inc. *Networking in Kubernetes*. [online] [cit. 2016-05-09]. Available from: <http://kubernetes.io/docs/admin/networking/>
- [16] Google, Inc. *Docker (Multi-Node)*. [online] [cit. 2016-05-09]. Available from: <http://kubernetes.io/docs/getting-started-guides/docker-multinode/>
- [17] Project Calico. *Calico for containers*. [online] [cit. 2016-05-09]. Available from: <https://github.com/projectcalico/calico-containers>
- [18] Project Calico. *Calico for containers*. [online] [cit. 2016-05-09]. Available from: <https://github.com/projectcalico/calico-containers/blob/v0.18.0/docs/cni/kubernetes/README.md>
- [19] CZ.NIC Labs. *The BIRD Internet Routing Daemon*. [online] [cit. 2016-05-09]. Available from: [http://bird.network.cz/?get\\_doc&f=bird-1.html](http://bird.network.cz/?get_doc&f=bird-1.html)
- [20] tcp cloud, a.s. *OpenContrail SDN Lab testing 1 – ToR Switches with OVSDB*. [online] [cit. 2016-05-09]. Available from: <http://www.tcpccloud.eu/en/blog/2015/07/13/opencontrail-sdn-lab-testing-1-tor-switches-ovsdb/>
- [21] Juniper Networks. *OpenContrail*. [online] [cit. 2016-05-09]. Available from: <http://www.opencontrail.org/opencontrail-quick-start-guide/>
- [22] Docker, Inc. *Docker Registry*. [online] [cit. 2016-05-09]. Available from: <https://docs.docker.com/registry/>
- [23] Docker, Inc. *Docker Registry Storage Driver*. [online] [cit. 2016-05-09]. Available from: <https://docs.docker.com/registry/storagedrivers/>
- [24] Google, Inc. *Kubernetes – Secrets*. [online] [cit. 2016-05-09]. Available from: <http://kubernetes.io/docs/user-guide/secrets/>
- [25] Treasure Data, Inc. *fluentd*. [online] [cit. 2016-05-09]. Available from: <http://www.fluentd.org/>
- [26] Elasticsearch BV. *Logstash*. [online] [cit. 2016-05-09]. Available from: <https://github.com/elastic/logstash/tree/2.2>

- [27] Mozilla Corporation. *Heka*. [online] [cit. 2016-05-09]. Available from: <https://hekad.readthedocs.org/en/latest/>
- [28] Shopify, Inc. *sarama*. [online] [cit. 2016-05-09]. Available from: <http://shopify.github.io/sarama/>
- [29] Prometheus Authors. *Prometheus*. [online] [cit. 2016-05-09]. Available from: <https://prometheus.io/docs/introduction/overview/>
- [30] Johnson, P. Don't go into programming if you don't have a good thesaurus. [online] [cit. 2016-05-09]. Available from: <http://www.itworld.com/article/2833265/cloud-computing/don-t-go-into-programming-if-you-don-t-have-a-good-thesaurus.html>
- [31] Wikipedia: the free encyclopedia. *Tarsier*. [online] [cit. 2016-05-09]. Available from: <https://en.wikipedia.org/wiki/Tarsier>
- [32] HashiCorp. *Consul*. [online] [cit. 2016-05-09]. Available from: <https://www.consul.io/>
- [33] Google, Inc. *The Go Programming Language*. [online] [cit. 2016-05-09]. Available from: <https://golang.org/>
- [34] Python Software Foundation. *python*. [online] [cit. 2016-05-09]. Available from: <https://www.python.org/>
- [35] Glozer, W. *wrk*. [online] [cit. 2016-05-09]. Available from: <https://github.com/wg/wrk>
- [36] Microsoft Corporation. Visio. [cit. 2016-05-09]. Available from: <https://products.office.com/cs-cz/visio/flowchart-software>
- [37] NGINX, Inc. *nginx*. [online] [cit. 2016-05-09]. Available from: <http://nginx.org/>
- [38] The Apache Software Foundation. Apache HTTP Server. [online] [cit. 2016-05-09]. Available from: <https://httpd.apache.org/>
- [39] Elasticsearch BV. Elasticsearch. [online] [cit. 2016-05-09]. Available from: <https://www.elastic.co/products/elasticsearch>
- [40] The Apache Software Foundation. HDFS Users Guide. [online] [cit. 2016-05-09]. Available from: [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_user\\_guide.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html)
- [41] Codenomicon Ltd. The Heartbleed Bug. [online] [cit. 2016-05-09]. Available from: <http://heartbleed.com/>

## BIBLIOGRAPHY

---

- [42] Software in the Public Interest, Inc. Debian. [online] [cit. 2016-05-09]. Available from: <https://www.debian.org/>
- [43] TROAN E., Brown P. logrotate. [online] [cit. 2016-05-09]. Available from: [http://www.linuxcommand.org/man\\_pages/logrotate8.html](http://www.linuxcommand.org/man_pages/logrotate8.html)
- [44] Red Hat, Inc. Project Atomic. [online] [cit. 2016-05-09]. Available from: <http://www.projectatomic.io/>
- [45] Y. Rekhter, Cisco Systems, B. Moskowitz, Chrysler Corp., D. Karrenberg, RIPE NCC, G. J. de Groot, RIPE NCC, E. Lear, Silicon Graphics, Inc. RFC 1918 – Address Allocation for Private Internets. [online] [cit. 2016-05-09]. Available from: <https://tools.ietf.org/html/rfc1918>
- [46] tcp cloud, a.s. tcp cloud. [online] [cit. 2016-05-09]. Available from: <http://www.tcpccloud.eu/>
- [47] Red Hat, Inc. Ansible. [online] [cit. 2016-05-09]. Available from: <https://www.ansible.com/>
- [48] Django Software Foundation. Django. [online] [cit. 2016-05-09]. Available from: <https://www.djangoproject.com/>
- [49] Ronacher, A. Flask (A Python Microframework). [online] [cit. 2016-05-09]. Available from: <http://flask.pocoo.org/>
- [50] Tornado Authors. Tornado Web Server. [online] [cit. 2016-05-09]. Available from: <http://www.tornadoweb.org/en/stable/>
- [51] Ronacher, A. Jinja2. [online] [cit. 2016-05-09]. Available from: <http://jinja.pocoo.org/docs/dev/>
- [52] Long, J. Python’s Web Framework Benchmarks. [online] [cit. 2016-05-09]. Available from: <http://klen.github.io/py-frameworks-bench/>
- [53] The Apache Software Foundation. Apache Kafka. [online] [cit. 2016-05-09]. Available from: <http://kafka.apache.org/>
- [54] Ruby community. Ruby. [online] [cit. 2016-05-09]. Available from: <https://www.ruby-lang.org/en/>
- [55] Node.js Foundation. Node.js. [online] [cit. 2016-05-09]. Available from: <https://nodejs.org/en/>
- [56] qwcode, dstufft, brosnier, carljm, jezdez, ianb. pip. [online] [cit. 2016-05-09]. Available from: <https://pip.pypa.io/en/stable/>
- [57] Elasticsearch BV. Kibana. [online] [cit. 2016-05-09]. Available from: <https://www.elastic.co/products/kibana>

- [58] Torkel Ödegaard & Coding Instinct AB. Grafana. [online] [cit. 2016-05-09]. Available from: <http://grafana.org/>
- [59] Evans, C. C. YAML. [online] [cit. 2016-05-09]. Available from: <http://yaml.org/>
- [60] . JSON. [online] [cit. 2016-05-09]. Available from: <http://www.json.org/>
- [61] Red Hat, Inc. GlusterFS. [online] [cit. 2016-05-09]. Available from: <https://www.gluster.org/>
- [62] Google, Inc. Goroutines. [online] [cit. 2016-05-09]. Available from: <https://tour.golang.org/concurrency/1>
- [63] Seznam.cz, a.s. DbgLog. [online] [cit. 2016-05-09]. Available from: <http://dbglog.sourceforge.net/>
- [64] @fat & @maccman. Bower. [online] [cit. 2016-05-09]. Available from: <http://bower.io/>
- [65] Farina, M. glide. [online] [cit. 2016-05-09]. Available from: <https://github.com/Masterminds/glide>
- [66] SwiftStack, Inc. OpenStack Swift. [online] [cit. 2016-05-09]. Available from: <https://www.swiftstack.com/openstack-swift/>
- [67] npm, Inc. npm. [online] [cit. 2016-05-09]. Available from: <https://www.npmjs.com/>
- [68] SaltStack, Inc. SaltStack. [online] [cit. 2016-05-09]. Available from: <http://saltstack.com/>





---

# Acronyms

<b>PaaS</b>	Platform as a service
<b>RC</b>	Replication Controller
<b>IP</b>	Internet Protocol
<b>API</b>	Application Programming Interface
<b>REST</b>	Representational State Transfer
<b>CPU</b>	Central processing unit
<b>HTML</b>	HyperText Markup Language
<b>CSS</b>	Cascading Style Sheets
<b>eth</b>	Ethernet network interface
<b>veth</b>	Virtual Ethernet device
<b>NAT</b>	Network Address Translation
<b>UDP</b>	User Datagram Protocol
<b>TCP</b>	Transmission Control Protocol
<b>TUN</b>	Network tunnel
<b>VM</b>	Virtual machine
<b>SDN</b>	Software-defined networking
<b>GET</b>	HTTP method
<b>POST</b>	HTTP method
<b>HTTP</b>	Hypertext Transfer Protocol

## A. ACRONYMS

---

**HTTPS** Hypertext Transfer Protocol Secure

**RAM** Random-access memory

**VLAN** Virtual LAN

**LAN** Local area network

**HDFS** Hadoop Distributed File System

**stdin** Standard input I/O connection

**stdout** Standard output I/O connection

**stderr** Standard error I/O connection

**I/O** input/output

**JVM** Java virtual machine

**SIGTERM** Generic signal used to cause program termination

**SIGCHLD** When a child process stops or terminates, SIGCHLD is sent to the parent process

**npm** Package manager for JavaScript

**GCC** GNU Compiler Collection

**GNU** GNU's Not Unix!

**PIP** Pip Installs Packages, Pip Installs Python

**CDN** Content delivery network

**YAML** YAML Ain't Markup Language

**JSON** JavaScript Object Notation

**FD** File descriptor

**SSL** Secure Sockets Layer

**URL** Uniform Resource Locator

---

## Contents of enclosed CD

readme.txt .....	the file with CD contents description
src .....	the directory of source codes
├─ go-kafkalog .....	Apache Kafka log format implementation in Go
├─ go-ultimate-server .....	Go server for benchmarking
├─ heka .....	Some Heka adjustment for greater reliability
├─ heka-kafkalog .....	Splitter and Decored plugins
├─ kafkafeeder .....	Application that transfers logs to Kafka
├─ kafkalog-logrus .....	Kafkalog hook to logrus
├─ kubernetes-testing .....	Support scripts for testing Kubernetes
├─ python-ultimate-server .....	Python server for benchmarking
├─ tarsier .....	Application fer testing Kubernetes cluster
text .....	the thesis text directory
├─ appendix .....	cd contents file and acronyms
├─ chapters .....	files of each chapter
├─ images .....	all used images
├─ inc .....	includes such as assignment, logo, class file, ...
├─ bibliography.bib .....	bibliography file
├─ sejvlond_masters_thesis.tex .....	main tex file
├─ sejvlond_masters_thesis.pdf .....	final PDF