Server Project Report

Boris Kirikov, Volodymir Lybinets, T5615SN, XAMK

Table of Contents

*All sources referenced in this report can be found on github:* [*k8s-lab-setup*](https://github.com/kribesk/k8s-lab-setup)*.*

# Introduction

The times when the software was targeted to a single machine unit have passed long ago when the very first commercial web service was introduced. The word web (i.e. network) brought user asymmetry so that the word “server” got its modern meaning – machine, serving one-to-many. The word commercial (i.e. business) called for reliability, redundancy and resiliency, so that the smallest downtime in the functionality of the service would mean lost profits.

With the growth of the web service popularity it proved that hosting one’s infrastruc-ture in one’s own premises is usually (not taking extreme cases into account) more expensive than renting the computational and storage volume from the company managing a datacenter, as the company provides servicing of multiple clients in bunch. Even more price reduction and client capacity was achieved with the applica-tion of hypervisors, which provided the possibility to run multiple virtual machines on a single hardware unit by dividing the physical resource capacity among them.

It was a good approach still widely used around the World’s datacenters, however, the load was growing rapidly and physical virtualization turned out to be incapable of solving two problems: ease of scalability for really large systems (those which re-quirements could not be satisfied by the capacity of one physical server) and load balancing.

And this is where container approach came into play. This approach takes into ac-count that most web services do not require separate OS environment and can be represented by conjunction of server software, configs, database and static data. Other than that, web services are mostly stateless, which means that no specific session information is stored in real time on the server between user requests, each request is fully processed “in single shot”. This gives the possibility of building the cluster infrastructure where the load is balanced over several machines containing replicas of containers on them so that each container could respond to user request.

And we are on this course to study exactly the latter approach.

## fabric8: continuous integration, code repository, build server and a lot more

Git is supposed to be used as code repository. It is simple, there are many plugins supporting hooks to notify building software on code updates. New code from the developers goes here, e.g. merging development branch with the master branch triggers CI server.

CI server performs three workflow steps: building application (compilation & link-age), launching unit-tests on a fresh build and, if a build had proven to be successful, wrapping it to the form of Docker container.

## Cluster management platform – orchestrator

This is the cluster management platform with the large bunch of functions. The most important for us include the examining of the state of cluster machines, deploying containers to them, balancing load over them and scheduling, which is the possibility for an administrator to specify how the certain container is run on a host machine.

Other than that, kubernetes provides the module named etcd which is responsible for storing shared configs of hosts and maintaining service information.

# Theoretical part

## Docker

*(Section by Boris Kirikov)*

**Docker** is a group of technologies and programs that introduce container, allow running containers, storing and managing them.

### Containers

Keeping service **installations inside VMs** become is very convenient way for deployment:

* If it works on developers machine, than it will work the same way everywhere
* Compilation, building, compilation, packaging, installation, configuration and other processes that form the way from development to production are simplified and done always in the same environment.
* Environment becomes part of the deployment
* Build systems and package managers often can help providing dependencies, but they are often limited to one platform (Java - Maven, Python - Pip, JS - Npm)
* And all the deploy is contained in one file: VM drive image

But VMs are typically rather big containing a lot of components for different use-cases: complete kernel, variety of drivers, libraries, user-space utils and basic services. This means a lot of overhead when packaging service or application with it’s own operating system.

This lead to idea of minimizing overhead of base images. Surely the base image only needs kernel, drivers used by VM solution, network services and management tools. Of cause, para-virtualization can be used. And so on.

The extreme of this idea is **OS-level virtualization** and containers. In this case there is no computation overhead and no extra kernel, because instances use directly the host’s kernel. Of cause host should support this feature.

The simplest implementation of this idea is UNIX chroot that basically just isolates fs sub-tree. So it is not a secure solution. Another classic solution is FreeBSD jail, that provides much more isolation and adds management features.

Nowadays Docker is very popular and widely supported solution for OS-level virtualization. And kubernetes can use docker as container driver.

### Images and containers

The two basic concepts of docker that should not be messed up are images and containers. An image is like virtual drive for VMs. It is file system. Container is like VM instance. It is application that has state and was booted from some container.

### Metadata

Docker (and kubernetes too) use “attached dictionaries” of metadata for simplifying and automation of management. This means that to objects (images, containers, volumes, networks, etc) labels can be attached. Label is basically a key-value pare of two strings, that can be tags, version/author information, scope or just everything.

For docker there are guidelines for using labels. Because docker images are way to distribute software and are mostly used by third-party users, namespace conventions for labels are suggested. They work like packages in java: reverse DNS notation of owned domain dot key name.

### Layered FS

One of the reasons why docker became so popular is that it works with containers like git with repositories. Instead of storing complete disk images it stores versions of images as difference layers. It minimizes use of storage and network, allows keeping different versions and quickly update images from registers.

The Docker storage driver stacks diff layers and provide a single unified view. When changes happen, they are written to the new layer on top of existing image.

### Registry and repository

When comparing docker to git, there are repositories and registries. Repository contains different versions in tree of layers of image, just like git contains code in tree of commits. Registry is a place that keeps repositories, just like GitHub. Examples of public registries are Docker Hub, Quay, GCR and AWS Container Registry. Just like git, docker enumerates versions with hashes and have tags pointing to specific versions.

### Docker daemon

Docker uses server-client model. Server is docker daemon, dockerd. And client may be a CLI application docker or compose (see later) etc. So dockerd is the service that runs containers. It provides REST API for communication with clients. There are different ways of running dockerd: as a standalone program, as systemd service and even as a windows service. Also different channels can be selected for communication: by default unix sockets are used, but tcp can also be used.

### Docker and Windows

Docker is designed for Linux based systems, but it is also available for Windows and Mac OS. This is possible because virtualization.

**Docker Machine** is a tool for installing and using Docker on virtual hosts. This does not limit to running Docker on other platforms but also allows to use it in data centers or with cloud providers. docker-machine CLI is used to install and control docker daemon on virtual hosts.

Modern way of installing Docker for Windows is not Docker Machine, but the idea is still the same. VM with Linux is created and CLI is used for hosting containers. The great difference is that dockerd is now windows native application, but still Linux kernel ([Alpine Linux](https://alpinelinux.org/) distribution aka MobyLinuxVM) is used. And Docker for Windows features nice installer and some GUI tools for configuration.

Also recently appeared option to run windows native containers. That means running windows applications inside native containers without any extra virtualization.

The only virtualization platform currently supported for Windows is Hyper-V. That means Hyper-V feature should be enabled on computer. To enable it:

1. Log in as administrator
2. Go to Control Panel -> Programs -> Turn Windows features on and off -> Enable Hyper-V Platform. Enabling management tools is not required but it simplifies troubleshooting process if something goes wrong
3. Reboot the computer (reboots twice)

After that Docker for Windows can be downloaded from official [download page](https://download.docker.com/win/stable/InstallDocker.msi) and installed. After that in Hyper-V management console new VM appears.

Docker allows to copy files (often configurations) to container’s RW layer. This is often convenient and needed to correctly run some containers. Because files should be copied from host’s FS to VM first, there should be file sharing enabled. This can be done in Docker GUI settings.

### Docker CLI

Basic commands are:

* docker pull <repository> to download repository from registry
* version or tag can be specified as <repository>:<tag>
* changing tag: docker tag <repository> <new\_repository>
* docker ps list containers
* docker rm <container> delete container
* docker images list images
* docker rmi <image> delete image
* docker run --name <container> <repository> run container from repository. Options:
  + -p <port>:<map> map port to local port
  + -v <path>:<path> map local file to container
  + -e <key>=<value> set environment variable
  + --network <net> connect to network
* docker attach <container> attach to running container
* docker exec -it <container> bash run bash in container
* docker logs <container> get logs for container
* docker stop <container> and docker <kill> container
* docker network ls to show networks
* docker network inspect <net> to get information about network
* docker network create --driver <driver> <net>

### Networking

Networking in docker is more complicated than it could be because of legacy features. Docker has built-in networks: bridge, host and none. And also allows user-defined networks: bridge, docker\_gwbridge and overlay. Connecting containers to none networks means running them without interfaces except loopback. Connecting to host network means using the same network as host.

Built-in bridge (interface docker0 of the host) and user-defined bridge create new virtual internal network, sets subnet and gateway.

Built-in bridge is a legacy bridge and it does not support discovery, but support links. Before networking there were to network mechanisms for containers: binding ports to host (exposing service) and links. Link create tunnel from one container to another and provides connection information with environmental variables and entry in hosts file.

User defined bridges do not support links. Containers added to network can immediately connect to other containers. Discovery for user defined bridges is provided by built-in DNS server.

docker\_gw network is created and connected automatically if container does not have external network/internet connectivity. overlay networks are used in swarm mode and overlay cluster network.

### Swarm

Swarm is another relatively-new feature of Docker that is basic cluster management. It’s list of features is quite similar to kubernetes’s, including: cluster networking and DNS, load balancers, scaling and self-maintenance according to declarative specifications.

Swarm allows to use cluster as one big docker engine, while Kubernetes has much more complex structure. Swarm has the same API as docker. Load balancing, grouping and exposing is done by another piece of software called Compose.

Detailed comparison of Swarm and Kubernetes can be found [here](https://platform9.com/blog/compare-kubernetes-vs-docker-swarm/).

### Example: running Keycloak

This example shows how to run Keycloak in docker container in MB316.

docker pull jboss/keycloak  
docker run --name keyserver -p 0.0.0.0:8081:8080 -d -e KEYCLOAK\_USER=admin -e KEYCLOAK\_PASSWORD=admin jboss/keycloak

The service will be available on http://localhost:8081/

## Packing application with docker

*(Section by Boris Kirikov)*

Packing application into docker image is easy. docker build command reads Dockerfile as a guide and creates image. This means some image is taken as a base and layer with application files is added as a new layer.

Syntax: docker build -t <image> . builds image using Dockerfile in current directory.

After that docker run --name <container> <image> can be used to start the app.

### Dockerfile

File consist of lines with instructions. Each line looks like <INSTRUCTION> <arguments>.

Basic instructions:

* FROM <image> use image as base
* RUN <command> run command in shell while building
* CMD <command> defines how to run container
* LABEL <key>=<value> adds label to image
* EXPOSE <port> exposes port
* ENV <key>=<value> set environment variable
* COPY <src> <dst> copy file to container
* USER <user> switches to user
* WORKDIR <path> sets working directory

### Example

Let’s create hello world web server using python.

1. mkdir hello-world && cd hello-world
2. cat > app.py this:

from bottle import route, run  
  
@route('/')  
def index():  
 return 'ok'  
  
run(host='0.0.0.0', port=13579)

1. cat > Dockerfile this:

FROM alpine:latest  
  
COPY app.py /src/app.py  
  
RUN apk add --update python py-pip  
RUN pip install bottle  
  
EXPOSE 13579  
CMD python /src/app.py

1. Now build: docker build -t hello-world .
2. And run: docker run --name hello-world-1 -p 13579:13579 -i -t hello-world
3. Browse to http://localhost:13579/

## Kubernetes

*(Section by Volodymyr Lubenets)*

### What is kubernetes (k8s)

Being formal, it is an open source container cluster manager. [wiki](https://en.wikipedia.org/wiki/Kubernetes)

Being descriptive, it is a platform, which takes care of managing the docker containers, which were previously decribed in this document, over the surface of cluster.

Management is a superclass-expression, which includes automatic or semi-automatic deployment of container VMs, scheduling (making the running environment i.e. cluster node correspond container requirements), failover management in case the container or a node goes down (self-healing) etc.

Also k8s is capable of managing the cluster itself, controlling it’s nodes, shared storages of different kind, shared configuration repository (see [etcd](https://github.com/coreos/etcd)) and also provide the UI for an administrator of the cluster.

Let’s take closer look on k8s’ features in corresponding section. As a summary, k8s is a platform merging a cluster manager, a container manager and a tool to control this systems.

### Who needs k8s and what for

Kubernetes is useful for the one who possesses eather physical (bare metal), virtual or mixed server park and wants to run a number of services (possibly large and fluctuating number) on the whole machine capacity with the ability to control and manage it simply.

### The key entities in the k8s architecture

The k8s whole infrastructure consists of two key entities, which provide the functonality of the system. Those are:

* Node

Nodes are the machines, either physical or virtual, which are capable of running at least: \* container runtime (docker) \* kubelet (a tool which monitors the state of the node and the pods on it and can change this states if desired) \* kube-proxy (an ip proxy and load balancer capable of routing the request/response to the correct node i.e. translating outer IP/port to cluster IP/port based on load balancing algorithm and vice versa) \* cAdvisor (CPU, memory and storage utilization monitor)

Cluster can consist of one to virtually unlimited number of nodes.

* Master

Master is a node which performs tasks of managing workload of the cluster and handling the direct communication to the it. There is usually one master node in the cluster and possibly several replicas if the high-availability cluster is implemented. Particularly, master runs such services, as:

* etcd (shared config repo)
* API server (HTTP server accepting JSONs containing commands)
* scheduler (a program which assignes pods to nods based on requirements of the former and load of the latter)
* controller manager (a process which the controllers (*see further*) run in)

### Entities running on top of infrastructure and controlling it

* Pod

This is higher-level abstraction over containers which ensures that the set of containers (the termin of *“the set”* should be understood *one or more* ) run in the shared environment which includes a single cluster-wide IP (to avoid port collisions over the cluster) and shared storage which can be initialized by pod. Single pod is run on a single node and is migrated as a whole to other node if required.

A pod is a basic structure being run on the cluster.

* Controller

This is a program which runs cyclically and drives the cluster from actual to desired state by adding/removing/moving/pausing/launching nodes, using API calls, to keep it simple. One can configure a custom controller, though there are several default ones:

* Replication controller, which ensures that a number of copies of the pod is being run over cluster
* DaemonSet controller, which is responsible for the particular pod to be run on each machine of the cluster in single instance
* JobController, which manages the pods which run to completion as part of batch jobs
* Service

The service is a group of pods that work together, which is entitled by a shared label and granted a cluster-wide persistant IP and the DNS record is added to cluster DNS db. This enables service exposure inside the cluster by default, but the service can also be exposed to the outside network if needed. Also the load balancer is activated over the service’s pod group.

### Useful links

* [API](https://kubernetes.io/docs/api/)
* [Design doc](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture.md)
* [CLI Commands](https://kubernetes.io/docs/user-guide/kubectl/v1.5/)
* [Wiki - kubernetes](https://en.wikipedia.org/wiki/Kubernetes)
* [Wiki - cluster manager](https://en.wikipedia.org/wiki/Computer_cluster#Cluster_management)

## Deploying docker container to kubernetes

*(Section by Volodymyr Lubenets,* ***TODO: Rewrite section!*** *)*

This is an example of deploying the service to kubernetes cluster. Let’s suppose the cluster is already up and running.

Other than that, suppose we have small python application, which acts as a HTTP server, as such. **app.py**

#!/usr/bin/python  
from BaseHTTPServer import BaseHTTPRequestHandler,HTTPServer  
  
PORT\_NUMBER = 8080  
  
#This class will handles any incoming request from  
#the browser   
class myHandler(BaseHTTPRequestHandler):  
   
#Handler for the GET requests  
def do\_GET(self):  
 self.send\_response(200)  
 self.send\_header('Content-type','text/html')  
 self.end\_headers()  
 # Send the html message  
 self.wfile.write("Hello World !")  
 return  
  
#Create a web server and define the handler to manage the  
#incoming request  
server = HTTPServer(('', PORT\_NUMBER), myHandler)  
  
#Wait forever for incoming http requests  
server.serve\_forever()

### Building docker container

The first thing to do is to write the provisioning script called Dockerfile. It contains the info on container’s base system, files to be inserted and commands to be executed. In here, the listing is such:

**Dockerfile**

FROM alpine:3.1  
  
COPY app.py /src/app.py  
   
RUN apk add --update python py-pip  
  
EXPOSE 13579  
CMD ["python", "/src/app.py"]

After this, execute the following commands:

docker build -t smallapp .  
docker run --name smallapp -1 -p 8080:13579 -i -t smallapp

### Deploying docker image to minikube

Run .\minikube.exe docker-env | Invoke-Expression (for powershell) to switch to minikube’s docker

Then run kubectl apply -f local-registry.yml

## Network boot techniques

*(Section by Boris Kirikov)*

When building cluster automating bare metal provisioning is absolutely essential. That means nodes can be provisioned with latest OS images with preinstalled cluster software, standard and custom configuration, booted. So that they’ll automatically join cluster with minimal amount of manual operations.

Our aim is to automate all the process after startup. Farther automation will require access to BIOS settings (like enabling Wake-On-LAN and boot device priority) and does not make that much simplification.

### PXE standard

PXE or Preboot eXecution Environment standard method for network boot. PXE is often a feature built-in network adapter. PXE is simple and very common: it is embedded in vast majority of modern network cards. Also MB316 computers have PXE feature.

PXE does not allow much freedom in boot process and used protocols, but it is still enough to boot everything. One of the reasons is that PXE like MBR have limit on size of firmware (32 KB).

Netboot is based on BOOTP/DHCP protocol:

1. Computer is started up and BIOS decides to initiate netboot
2. Control is passed to PXE firmware of network card
3. Network card tries to get configuration from DHCP server
4. If there is DHCP server on the network, network card gets configuration
5. In addition to settings like IP, mask, gateway and DNS, DHCP protocol can provide number of additional fields and “vendor specific fields”
6. The interesting fields are filename and next-server
7. PXE firmware connects via TFTP protocol to next-server, downloads filename and executes it

This means that to set up netboot we need to start reconfigure DHCP server and start TFTP server on our network.

The next file that is executed (filename) can be everything, it is not limited to OS image. For example it can be more powerful netboot software that will allow perform more sophisticated boot process. Examples of such software are pxelinux, GRUB and iPXE.

### iPXE

“iPXE is the leading open source network boot firmware. It provides a full PXE implementation enhanced with additional features”.

That means it is still PXE and operates in the same way. If we will use it like that, iPXE will connect to DHCP, get configuration, download and iPXE again. And again, and again. This loop should be broken. There are two options for that: set up DHCP server so that it it provides different configuration for iPXE clients or modify iPXE (rebuild with embedded script) so that it won’t boot from DHCP settings.

### iPXE as a shell

Firmware implementations of PXE are straightforward: they just perform described algorithm. But iPXE is different. iPXE has shell and scripting language that allows customizing boot process, using different protocols and displaying menus.

iPXE script can be provided by DHCP server in filename field. Now this address can be not on TFTP server, but it can be web server. More generally, it can be not a static file but generated by server in response to iPXE request.

iPXE script start with shebang #!ipxe and after that come lines of commands. The most important command is chain <file> that chainloads to specified file, that can be accessed through supported protocol, like HTTP, FTP, iSCSI, FCoE and others. dhcp is used to contact DHCP server and obtain configuration. Scripting allows also comments, flow control with goto and short-circuit logic and variables. Full list of command available in [command reference](http://ipxe.org/cmd).

So the main idea is that iPXE with good boot script can boot everything we need.

### Matchbox

Matchbox is set of tools and services for netboot and provision of CoreOS, host OS for kubernetes.

Matchbox services are packed into docker (and rkt) containers. The main service provides iPXE http server, that generates iPXE boot scripts and also ignition provision configuration for CoreOS (described in CoreOS section) and metadata. That is everything we need to boot from iPXE.

For PXE stage, matchbox suggests using dnsmasq simple dhcp/dns/tftp server packed also into docker container.

As soon as both used services are packed into containers, they can run inside cluster after first nodes are provisioned.

Matchbox configuration is based on groups and profiles. If node matches group’s selector, group profile is used to build configuration files. Profile defines which images are used to boot host and what config templates are used to generate configuration files.

So the overall process is:

1. Host is powered on, BIOS performed checks and passes control to network card’s PXE
2. PXE requests configuration from DHCP dnsmasq server (running in standalone docker or already in cluster)
3. Server determines that it is not iPXE and returns its address as next-server and undionly.kpxe as filename
4. PXE loads iPXE from dnsmasq tftp and runs it
5. iPXE requests configuration from DHCP dnsmasq server
6. Server determines that it is now iPXE and returns address of matchbox iPXE endpoint: filename = http://<matchbox>:8080/ipxe?mac=${mac}
7. iPXE downloads script and chains to it
8. Script provide address of image and additional parameters, like ignition config url
9. CoreOS image is downloaded, control is passed to CoreOS ignition system
10. Ignition config is downloaded and used to correctly boot CoreOS, set everything up and connect to cluster (or start it, if belongs to controller group in matchbox config)

## CoreOS

*(Section by Volodymyr Lubenets)*

### What is it?

CoreOS is a linux distribution adapted to be used as an OS for a cluster node, particularly the cluster managed by kubernetes platforms and using docker container approach.

CoreOS distribution differs from classical desktop or server dramatically. It possesses special features which make OS-level virtualization easy and possible. Let’s take a look on them.

### Special features of CoreOS

* No package manager used. CoreOS is designed to just provide its kernel to virtualized containers, so the native software set of the distribution is not supposed to be dynamically modified.
* Uses rkt as built-in container runtime. This is a linux native container runtime developed by CoreOS team, which uses linux kernel instruments such as cgroups and namespaces instead of Docker approach (btw, it was used here previously too)
* Has a special built-in daemon etcd, which was described in *“Kubernetes”* section. Keeping short, it is a program which gets the configs from cluster-wide repo and can write to the shared cluster storage dynamically. Other than that, etcd is responsible for service discovery feature allowing other cluster members see and use the service running.
* etcd also provides the HTTP API server which gets JSONs with commands from the orchestrator or can be also managed by its etcdctl
* Ignition and ignition configs. This is a feature which performs the kick-off configuration of the system on its boot. It is really the distinctive feature of CoreOS, so more info - in the section below.

### Ignition

The special thing about ignition is that it is executed in the early boot even before initramfs, so it is capable of doing such things, as:

* partitioning disks
* formatting partitions
* configuring users
* writing files
* writing network configuration

etc.

So, the point of using ignition is that you don’t need to build each distribution for each cluster machine preconfigured and you may make a server which will distribute the initial configurations based on IP/hostname of the machine connecting. This can reduce the time of getting cluster to work significantly.

[More information](https://coreos.com/ignition/docs/latest/what-is-ignition.html) on ignition.

### Supported platforms

With CoreOS, you can build a cluster on practically any computational unit you have, disregarding is that bare metal device, VM possessed by Amazon, Google, or your on-site infrastructure or some dynamic number of machines connecting over the internet (PXE boot).

Every platform will function the same way and it supports ignition configs.

The whole list of platforms can be found [here](https://coreos.com/ignition/docs/0.12.1/supported-platforms.html).

# Notes on implementing

## Trying minikube on Windows

*(Section by )*

…

## Starting matchbox on Windows host

*(Section by )*

### Why can not use native Docker

…

### 

## Preparing configs

*(Section by )*

### Lab address schema

…used ips/mask/MACs, dns records, node names etc

### Matchbox configs schema

…how matchbox can build custom configs using profiles and groups… …what is schema our configs: give tree configs/ output and describe files and how they go together…

### Config generator utility

…describe our tool for generating ip/MAC independent configs…

### WOL, SSH, shutdown helpers

…about out helper tools: wake.cmd, ssh-node.cmd, shutdown.cmd…

## Other notes

*(Section by )*

### K8s persistent storage

…using nfs…

### Exposing services is not so simple

…service types: clusterip, nodeport and external balances…

…new way/our way: ingresses with load balancer in container…

# Result description

## Project Readme

…the same as README.md…

# References

[1] “Wikipedia.” [Online]. Available: <https://en.wikipedia.org/>.