

Politechnika Warszawska

W Y D Z I A Ł E L E K T R Y C Z N Y



Instytut Elektrotechniki Teoretycznej
i Systemów Informacyjno-Pomiarowych
Zakład Elektrotechniki Teoretycznej
i Informatyki Stosowanej

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria oprogramowania

Implementacja środowiska Kubernetes na maszynach
bezdyskowych

Krzysztof Nazarewski

nr albumu 240579

promotor
mgr inż. Andrzej Toboła

WARSZAWA 2018

Implementacja środowiska Kubernetes na maszynach bezdyskowych

Streszczenie

Celem tej pracy inżynierskiej jest przybliżenie czytelnikowi zagadnień związanych z systemem Kubernetes oraz jego uruchamianiem na maszynach bezdyskowych.

Zacznę od wyjaśnienia pojęcia kontenerów, problemu orkiestracji nimi i krótkiego teoretycznego przeglądu dostępnych rozwiązań. Opiszę czym jest Kubernetes, jaką ma architekturę oraz przedstawię podstawowe pojęcia pozwalające na zrozumienie i korzystanie z niego. Opis Kubernetes zakończę przedstawieniem sposobów jego uruchomienia na maszynach bezdyskowych.

Następnie przeprowadzę krótki teoretyczno-praktyczny przegląd systemów operacyjnych i sposobów uruchamiania Kubernetes na nich.

Po ich wybraniu przeprowadzę testy na sieci uczelnianej, a na koniec doprowadzę ją do stanu docelowego pozwalającego na przeprowadzenie laboratoriów Kubernetes.

Słowa kluczowe: Kubernetes, konteneryzacja, orkiestracja, maszyny bezdyskowe

Implementing Kubernetes on diskless machines

Abstract

Primary goal of this document is to present basic concepts related to Kubernetes and running it on diskless systems.

I will start with explaining what are containers, problem of their orchestration and I will theoretically inspect available solutions.

I will describe what is Kubernetes, provide overview of its architecture and basic concepts allowing to understand and use it. I will end the description with overview of its provisioning tools working on diskless systems.

Then I will research and conduct brief practical tests of available operating systems and provisioning tools.

After selecting solutions I will conduct practical tests on university network and finally configure Kubernetes to run the network.

Keywords: Kubernetes, containerization, orchestration, diskless systems

WARSZAWA, 1 lutego 2018

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa inżynierska pt. Implementacja środowiska Kubernetes na maszynach bezdyskowych:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Krzysztof Nazarewski.....

Spis treści

1	Wstęp	1
1.1	Konteneryzacja	3
1.2	Cel pracy inżynierskiej	4
2	Przegląd pojęć i systemów związanych z konteneryzacją	5
2.1	Open Container Initiative	5
2.2	<i>Docker</i>	5
2.3	Dostępne rozwiązania zarządzania kontenerami	6
3	<i>Kubernetes</i>	8
3.1	Administracja, a korzystanie z klastra	8
3.2	Konfiguracja klastra	8
3.3	Infrastruktura <i>Kubernetesa</i>	9
3.4	Architektura	12
3.5	<i>Kubernetes Dashboard</i>	16
3.6	<i>Kubernetes Incubator</i>	16
3.7	Administracja klastrem z linii komend	17
3.8	Administracja klastrem za pomocą narzędzi graficznych	18
3.9	Lista materiałów dodatkowych	19
4	Systemy bezdyskowe	20
4.1	Proces uruchamiania maszyny bezdyskowej	20
5	Przegląd systemów operacyjnych	22
5.1	Konfigurator cloud-init	22
5.2	<i>CoreOS</i>	23
5.3	RancherOS	24
5.4	<i>Project Atomic</i>	25
5.5	Alpine Linux	25
5.6	ClearLinux	25
5.7	Wnioski	26

6	Praktyczne rozeznanie w narzędziach administracji klastrem	27
	<i>Kubernetesa</i>	
6.1	kubescape-cli	27
6.2	Rancher 2.0	28
6.3	OpenShift Origin	31
6.4	kubescape	34
6.5	Wnioski	35
7	Uruchamianie <i>Kubernetesa</i> w laboratorium 225	36
7.1	Przygotowanie węzłów <i>CoreOS</i>	36
7.2	Przeszkody związane z uruchamianiem skryptów na uczelnianym Ubuntu	37
7.3	Pierwszy dzień - uruchamianie skryptów z maszyny s6	38
7.4	Kolejne próby uruchamiania klastra z maszyny s2	39
8	Docelowa konfiguracja w sieci uczelnianej	44
8.1	Procedura uruchomienia klastra	44
8.2	Sprawdzanie, czy klastr działa	47
9	Rezultaty i wnioski	51
A	Wykaz skryptów	52
A.1	ipxe-boot	52
A.2	kubernetes-cluster	52

Rozdział 1

Wstęp

W ostatnich latach na popularności zyskują tematy związane z izolacją aplikacji, konteneryzacją i zarządzaniem rozproszonymi systemami komputerowymi.

Sam problem izolacji systemów komputerowych istnieje już od dawna i dorobił się wielu podejść do jego rozwiązania:

- rozwijane od późnych lat 60 wirtualne maszyny dzielące się na dwa rodzaje:
 - systemowe lub inaczej emulatory maszyn, w uproszczeniu polegają na uruchamianiu kompletnego systemu operacyjnego, który nie zdaje sobie sprawy ze współdzielenia zasobów “myśląc”, że posiada całą fizyczną maszynę na własność. Możliwe jest uruchomienie całkiem innego systemu operacyjnego jako gościa;
 - działające na poziomie procesu; oferują przede wszystkim izolację zależności i niezależność od systemu operacyjnego, można tu wyróżnić między innymi:
 - * interpretery (np. *CPython* lub *Lua*),
 - * kompilatory JIT (np. *Jython*, *PyPy*, *LuaJIT*, *.NET*),
 - * maszyny wirtualne języków programowania (np. *Java* lub *V8*);
- wprowadzony w roku 1979 *chroot* polegający na uruchomieniu procesu ze zmienionym drzewem systemu plików, z którego nie może się wydostać;
- parawirtualizacja, która jest podobna do systemowych wirtualnych maszyn, z tą różnicą, że przekierowuje zapytania systemowe do systemu gospodarza. Ten typ wirtualizacji nie pozwala na uruchomienie całkiem innego systemu operacyjnego i wymaga kompatybilności systemu-

gościa. Przykładem jej implementacji są *jail* z *FreeBSD* lub *LXC* (*Linux Containers*).

1.1 Konteneryzacja

Pełna wirtualizacja systemów operacyjnych świetnie się sprawdza przy współdzieleniu zasobów sprzętowych z niezaufanymi użytkownikami. Na przykład w centrach danych lub usługach chmurowych.

Natomiast rozwój realnych aplikacji i usług internetowych dąży do izolacji jak najmniejszych ich części na poziomie pojedynczego procesu. Niektórzy idą dalej i rozbijają procesy na jeszcze mniejsze jednostki (tzw. mikroserwisy) ograniczając ich funkcjonalność do minimum.

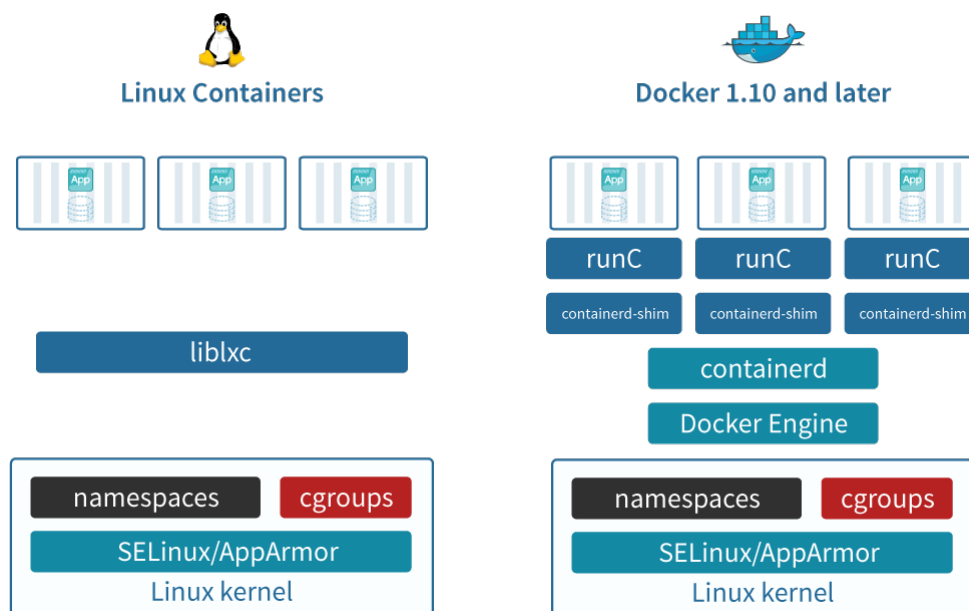
Zastosowanie w tym przypadku pełnej wirtualizacji skutkowałoby nieproporcjonalnie dużym narzutem zasobów sprzętowych, a przez to finansowych, w stosunku do uruchamianej aplikacji. Standardowe narzędzia parawirtualizacji zmniejszają ten narzut, ale nadal jest znaczny i wymaga dalszej optymalizacji.

W ten sposób zrodziła się idea konteneryzacji. Polega ona na:

- uruchamianiu pojedynczych procesów,
- działaniu we w pełni skonfigurowanym środowisku niezależnym od innych procesów współdzielących system operacyjny,
- dążeniu do minimalizacji kosztów uruchamiania kolejnych procesów.

Warto tu zaznaczyć, że konteneryzacja nie jest jedynym narzędziem lub gotowym rozwiązaniem. Jest natomiast dobrze określonym zbiorem problemów i recept na ich rozwiązanie.

Konteneryzację realizuje się łącząc wiele istniejących lub nowych narzędzi optymalizowanych w konkretnym celu. Sytuację dobrze ilustruje poniższe porównanie *LXC* z *Dockerem*:



Jak widać na powyższej ilustracji zarówno *LXC* jak i *Docker* bazują na kernelu *Linuxa*, w tym: *SELinux* lub *AppArmor*, *namespaces* i *cGroups*. Różnią się natomiast implementacją samych kontenerów - *LXC* korzysta jedynie z *liblxc*, a *Docker* postanowił zaimplementować wielopoziomowy system: *Docker Engine*, *containerd* i *runc*.

1.2 Cel pracy inżynierskiej

Celem tej pracy jest: 1) przedstawienie podstawowych pojęć związanych z aktualnie najpopularniejszym rozwiązaniem zarządzania kontenerami o nazwie *Kubernetes*, 2) przegląd dostępnych rozwiązań oraz wdrożenie tego systemu w sieci uczelnianej na potrzeby prowadzenia laboratoriów ze studentami.

Wdrożenie w sieci uczelnianej wiąże się z koniecznością uruchomienia systemu z sieci na maszynach bezdyskowych.

Celem dodatkowym jest przeprowadzenie testów wydajnościowych klastra *Kubernetes*.

Rozdział 2

Przegląd pojęć i systemów związanych z konteneryzacją

Wiodącym, ale nie jedynym, rozwiązaniem konteneryzacji jest *Docker*.

2.1 Open Container Initiative

Open Container Initiative jest inicjatywą tworzenia i utrzymywania publicznych standardów związanych z tworzeniem i obsługą kontenerów.

Większość projektów związanych z konteneryzacją dąży do kompatybilności ze standardami *OCI*, m. in.:

- *Docker*
- *Kubernetes CRI-O*
- *Docker on FreeBSD*
- *Running CloudABI applications on a FreeBSD based Kubernetes cluster*, by Ed Schouten (*EuroBSDcon '17*)

2.2 Docker

Docker jest najstarszym i w związku z tym aktualnie najpopularniejszym rozwiązaniem problemu konteneryzacji.

Dobrym przeglądem alternatyw dla *Dockera* jest porównanie *rkt* z innymi rozwiązaniami na oficjalnej stronie *CoreOS*.

Domyślnie obrazy są pobierane przez internet z *Docker Hub*a, co jest ograniczone przepustowością łącza. Na wolne łącze możemy zaradzić kieszonując zapytania HTTP lub uruchamiając rejestr obrazów w sieci lokalnej.

Lokalny rejestr może być ograniczony do obrazów ręcznie w nim umieszczonych lub udostępniać i kieszeniować obrazy z zewnętrznego rejestru (np. *Docker Hub*). Pierwsze rozwiązanie w połączeniu z zablokowaniem dostępu do zewnętrznych rejestrów daje prawie pełną kontrolę nad obrazami uruchamianymi wewnątrz sieci.

2.3 Dostępne rozwiązania zarządzania kontenerami

Kubernetes

Kubernetes (w skrócie *k8s*) jest obecnie najpopularniejszym narzędziem orkiestracji kontenerami, a przez to tematem przewodnim tego dokumentu.

Został stworzony przez *Google* na bazie ich wewnętrznego systemu Borg.

W porównaniu do innych narzędzi *Kubernetes* oferuje najlepszy kompromis między oferowanymi możliwościami, a kosztem zarządzania.

Dobrym przeglądem alternatyw *Kubernetes* jest artykuł pt. *Choosing the Right Containerization and Cluster Management Tool*.

Fleet

Fleet jest nakładką na *systemd* realizująca rozproszony system inicjalizacji systemów w systemie operacyjnym *CoreOS*.

Kontenery są uruchamiane i zarządzane przez *systemd*, a stan przechowywany jest w *etcd*.

Aktualnie projekt kończy swój żywot na rzecz *Kubernetes* i w dniu 1 lutego 2018, został wycofany z domyślnej dystrybucji *CoreOS*. Nadal będzie dostępny w rejestrze pakietów *CoreOS*.

Docker Swarm

Docker Swarm jest rozwiązaniem orkiestracji kontenerami od twórców samego *Dockera*. Proste w obsłudze, ale nie oferuje tak dużych możliwości jak inne rozwiązania.

Nomad

Nomad od *HashiCorp* jest narzędziem do zarządzania klastrem, które również oferuje zarządzanie kontenerami.

Przy jego tworzeniu twórcy kierują się filozofią *Unix*. W związku z tym Nomad jest prosty w obsłudze, wyspecjalizowany i rozszerzalny. Zwykle działa w tandemie z innymi produktami HashiCorp jak Consul i Vault.

Porównanie z innymi rozwiązaniami możemy znaleźć na oficjalnej stronie *Nomada: HashiCorp Nomad vs Other Software*

Mesos

Apache Mesos jest najbardziej zaawansowanym i najlepiej skalującym się rozwiązaniem orkiestracji kontenerami. Jest również najbardziej skomplikowanym i trudnym w zarządzaniu rozwiązaniem, w związku z tym znajduje swoje zastosowanie tylko w największych systemach komputerowych.

Dobrym wstępem do zagadnienia jest ten artykuł: *An Introduction to Mesosphere*.

Rozdział 3

Kubernetes

W tym rozdziale opiszę zagadnienia związane z samym systemem *Kubernetes*.

3.1 Administracja, a korzystanie z klastra

Przez zwrot *administracja klastrem* (lub zarządzanie nim) rozumiem zbiór czynności i procesów polegających na przygotowaniu klastra do użytku i zarządzanie jego infrastrukturą. Na przykład: tworzenie klastra, dodawanie i usuwanie węzłów oraz nadawanie uprawnień innym użytkownikom.

Przez zwrot *korzystanie z klastra* rozumiem uruchamianie aplikacji na działającym klastrze.

Ze względu na ograniczone zasoby czasu w tej pracy inżynierskiej skupiam się na kwestiach związanych z administracją klastrem.

3.2 Konfiguracja klastra

Ważną kwestią jest zrozumienie pojęcia stanu w klastrze *Kubernetes*. Jest to stan do którego klaster dąży, a nie w jakim się w danej chwili znajduje.

Zwykle stan docelowy i aktywny szybko się ze sobą zbiegają, ale nie jest to regułą. Najczęstszymi scenariuszami jest brak zasobów do uruchomienia aplikacji w klastrze lub zniknięcie węzła roboczego.

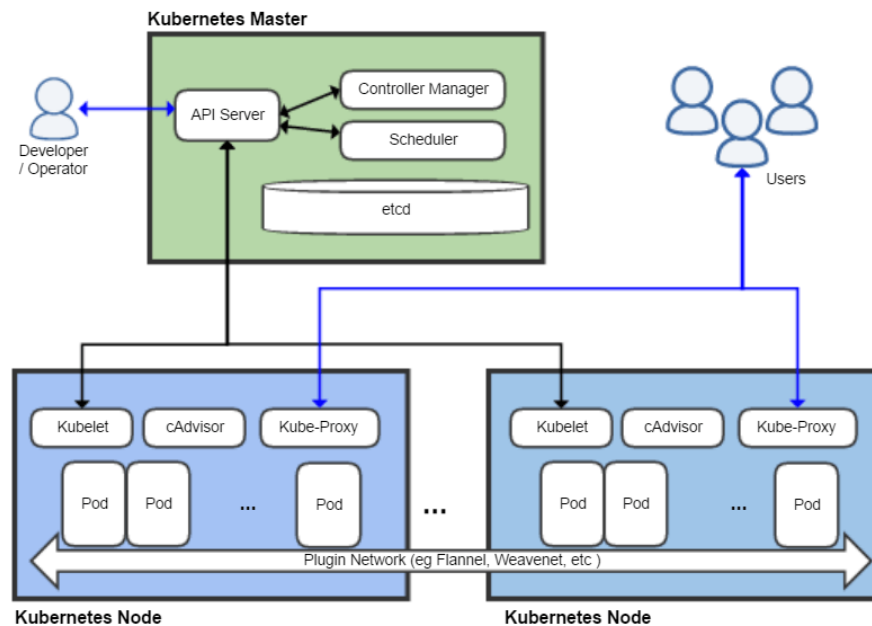
W pierwszym przypadku stan klastra może wskazywać na istnienie 5 instancji aplikacji, ale pamięci RAM wystarcza na uruchomienie tylko 3. Więc bez zmiany infrastruktury brakujące 2 instancje nigdy nie zostaną uruchomione. W momencie dołączenia kolejnego węzła klastra może się okazać, że posiadamy już oczekiwane zasoby i aplikacja zostanie uruchomiona w pełni.

W drugim przypadku założymy, że aplikacja jest uruchomiona w 9 kopiach na 4 węzłach, po 2 kopie na pierwszych trzech węzłach i 3 kopie na ostatnim. W momencie wyłączenia ostatniego węzła aplikacja będzie miała uruchomione tylko 6 z 9 docelowych instancji. Zanim moduł kontrolujący klastera zauważy braki aktywny stan 6 nie będzie się zgadzał z docelowym 9. W ciągu kilku do kilkudziesięciu sekund kontroler uruchomi brakujące 3 instancje i uzyskamy docelowy stan klastra: po 3 kopie aplikacji na 3 węzłach.

3.3 Infrastruktura *Kubernetesa*

Infrastrukturę definiuję jako część odpowiadającą za funkcjonowanie klastra, a nie za aplikacje na nim działające. Z infrastrukturą wiąże pojęcie administracji klastrem.

Zdecydowałem się przybliżyć temat na podstawie jednego diagramu znalezionej na wikimedia.org:



Na ilustracji możemy wyróżnić 5 grup funkcjonalnych:

1. *Developer/Operator*, czyli administrator lub programista korzystający z klastra,
2. *Users*, czyli użytkowników aplikacji działających w klastrze,
3. *Kubernetes Master*, czyli węzeł zarządzający (zwykle więcej niż jeden),

4. *Kubernetes Node*, czyli jeden z wielu węzłów roboczych, na których działają aplikacje,
5. *Plugin Network*, czyli wtyczka sieciowa realizująca lub konfiguruje połączenia pomiędzy kontenerami działającymi w ramach klastra,

Węzeł zarządzający

Stan *Kubernetes* jest przechowywany w *etcd*. Nazwa wzięła się od Unixowego folderu */etc* przechowującego konfigurację systemu operacyjnego i litery *d* oznaczającej system rozproszony (ang. distributed system). Jest to baza danych przechowująca jedynie klucze i wartości (ang. key-value store). Koncepcyjnie jest prosta, żeby umożliwić skupienie się na jej wydajności, stabilności i skalowaniu.

Jedynym sposobem zmiany stanu *etcd* (zakładając, że nie jest wykorzystywane do innych celów) jest komunikacja z *kube-apiserver*. Zarówno zewnątrzni użytkownicy jak i wewnętrzne procesy klastra korzystają z interfejsu aplikacyjnego REST (ang. REST API) klastra w celu uzyskania informacji i zmiany jego stanu.

Głównym modulem zarządzającym, który dba o doprowadzenia klastra do oczekiwanego stanu jest *kube-controller-manager*. Uruchamia on pętle kontrolujące klaster, na której bazuje wiele procesów kontrolnych jak na przykład kontroler replikacji i kontroler kont serwisowych.

Modulem zarządzającym zasobami klastra jest *kube-scheduler*. Decyduje on na których węzłach uruchamiać aplikacje, żeby zaspokoić popyt na zasoby jednocześnie nie przeciążając pojedynczych węzłów klastra.

Węzeł roboczy

Podstawowym procesem działającym na węzłach roboczych jest *kubelet*. Monitoruje i kontroluje kontenery działające w ramach jednego węzła. Na przykład wiedząc, że na węźle mają działać 2 instancje aplikacji dba o to, żeby restartować instancje działające nieprawidłowo i/lub dodawać nowe.

Drugim najważniejszym procesem węzła roboczego jest *kube-proxy* odpowiadające za przekierowywanie ruchu sieciowego do odpowiednich kontenerów w ramach klastra.

Ostatnim opcjonalnym elementem węzła roboczego jest *cAdvisor* (Container Advisor), który monitoruje zużycie zasobów i wydajność kontenerów w ramach jednego klastra.

Wtyczka sieciowa

Podstawowym założeniem *Kubernetesa* jest posiadanie własnego adresu IP przez każdą aplikację działającą w klastrze, ale nie narzuca żadnego rozwiązania je realizującego.

Administrator (lub skrypt konfigurujący) klastra musi zadbać o to, żeby skonfigurować wtyczkę sieciową realizującą to założenie.

Najprostszym koncepcyjnie rozwiązaniem jest stworzenie na każdym węźle wpisów *iptables* przekierowujących adresy IP na wszystkie inne węzły.

Jednymi z najpopularniejszymi rozwiązaniami są: Flannel i Project Calico.

Komunikacja sieciowa

Materiały źródłowe:

- <https://www.slideshare.net/weaveworks/kubernetes-networking-78049891>
- <https://jvns.ca/blog/2016/12/22/container-networking/>
- https://medium.com/@anne_e_currie/kubernetes-aws-networking-for-dummies-like-me-b6dedeeb95f3

4 rodzaje komunikacji sieciowej:

1. wewnątrz Podów (localhost)
2. między Podami (trasowanie lub nakładka sieciowa - overlay network)
3. między Podami i Serwisami (kube-proxy)
4. świata z Serwisami

W skrócie:

- *Kubernetes* uruchamia *Pody*, które implementują Serwisy,
- *Pody* potrzebują *Sieci Podów* - trasowanych lub nakładkę sieciową,
- *Sieć Podów* jest sterowana przez *CNI* (Container Network Interface),
- Klient łączy się do Serwisów przez wirtualne IP Klastra,
- *Kubernetes* ma wiele sposobów na wystawienie Serwisów poza klaster,

Zarządzanie dostępami

Podstawowymi pojęciami związanymi z zarządzaniem dostępami w *Kubernetesie* są uwierzytelnianie, autoryzacja i *Namespace*.

Uwierzytelnianie Pierwszym krokiem w każdym zapytaniu do API jest uwierzytelnienie, czyli weryfikacja, że użytkownik (czy to aplikacja) jest tym za kogo się podaje. Podstawowymi sposobami uwierzytelniania są:

- certyfikaty klienckie X509,
- statyczne przepustki (ang. *token*),
- przepustki rozruchowe (ang. *bootstrap tokens*),
- statyczny plik z hasłami,
- przepustki kont serwisowych (ang. *ServiceAccount tokens*),
- przepustki OpenID Connect,
- Webhook (zapytanie uwierzytelniające do zewnętrznego serwisu),
- proxy uwierzytelniające,

Ze względu na prostotę i uniwersalność rozwiązania w tej pracy będę korzystał z *ServiceAccount*.

Autoryzacja Drugim krokiem jest autoryzacja, czyli weryfikacja, że użytkownik jest uprawniony do korzystania z danego zasobu.

Najpopularniejszym sposobem autoryzacji jest RBAC (Role Based Access Control). Odbywa się ona na podstawie ról (*Role* i *ClusterRole*), które nadają uprawnienia i są przypisywane konkretnym użytkownikom lub kontom przez *RoleBinding* i *ClusterRoleBinding*.

Namespace (przestrzeń nazw) jest logicznie odseparowaną częścią klastra *Kubernetes*. Pozwala na współdzielenie jednego klastra przez wielu niezauważanych użytkowników. Standardowym zastosowaniem jest wydzielanie środowisk produkcyjnych, QA i deweloperskich.

Jak nazwa wskazuje role z dopiskiem *Cluster* mogą dać dostęp do wszystkich przestrzeni nazw jednocześnie oraz zasobów takowych nie posiadających. Przykładem zasobu nie posiadającego swojej przestrzeni nazw jest węzeł (*Node*) lub zakończenie API */healthz*.

Role bez dopisku *Cluster* operują w ramach jednej przestrzeni nazw.

3.4 Architektura

Architekturę klastra definiuję jako część aplikacyjną, czyli wszystkie funkcjonalności dostępne po przeprowadzeniu prawidłowej konfiguracji klastra i oddaniu węzłów do użytku. Z architekturą wiąże pojęcia korzystania z klastra, stanu i obiektów *Kubernetesa*.

Obiekty Kubernetes API

Obiekty Kubernetesa są trwale przechowywane w *etcd* i definiują, jak wcześniej wyjaśniłem, pożądany stan klastra. Szczegółowy opis konwencji API obiektów możemy znaleźć w odnośniku.

Jako użytkownicy klastra operujemy na ich reprezentacji w formacie YAML, a rzadziej JSON, na przykład:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: my-pod
5   namespace: my-namespace
6   uid: 343fc305-c854-44d0-9085-baed8965e0a9
7   labels:
8     resources: high
9   annotations:
10    app-type: qwe
11 spec:
12   containers:
13   - image: ubuntu:trusty
14     command: ["echo"]
15     args: ["Hello World"]
16   ...
17 status:
18   podIP: 127.12.13.14
19   ...
```

W każdym obiekcie możemy wyróżnić trzy obowiązkowe i dwa opcjonalne pola:

- *apiVersion*: obowiązkowa wersja API *Kubernetes*,
- *kind*: obowiązkowy typ obiektu zdefiniowanego w specyfikacji *apiVersion*,
- *metadata*
 - *namespace*: opcjonalna (domyślna *default*) przestrzeń nazw do której należy obiekt,
 - *name*: obowiązkowa i unikalna w ramach przestrzeni nazw nazwa obiektu,
 - *uid*: unikalny identyfikator obiektu tylko do odczytu,
 - *labels*: opcjonalny zbiór kluczy i wartości ułatwiających identyfikację i grupowanie obiektów,
 - *annotations*: opcjonalny zbiór kluczy i wartości wykorzystywanych przez zewnętrzne lub własne narzędzia,

- *spec*: z definicji opcjonalna, ale zwykle wymagana specyfikacja obiektu wpływająca na jego funkcjonowanie,
- *status*: opcjonalny aktualny stan obiektu tylko do odczytu,

Podstawowe rodzaje obiektów aplikacyjnych

Ważną kwestią jest rozróżnienie obiektów imperatywnych i deklaratywnych. Obiekty imperatywne reprezentują wykonanie akcji, a deklaratywne określają stan w jakim klaster powinien się znaleźć.

Pod *Pod* jest najmniejszą jednostką aplikacyjną w *Kubernetesie*. Reprezentuje nierozłącznie powiazaną (np. współdzielonymi zasobami) grupę jednego lub więcej kontenerów.

Pod w odróżnieniu od innych obiektów reprezentuje aktualnie działającą aplikację. Są bezustannie uruchamiane i wyłączane przez kontrolery. Trwałość danych można uzyskać jedynie przydzielając im zasoby dyskowe.

Pody nie powinny być zarządzane bezpośrednio, jedynie przez kontrolery. Najczęściej konfigurowane są przez *PodTemplateSpec*, czyli szablony ich specyfikacji.

Kontenery wewnątrz *Poda* współdzielą adres IP i mogą komunikować się przez *localhost* i standardowe metody komunikacji międzyprocesowej.

Dodatkowo kontenery wewnątrz *Podów* obsługują 2 rodzaje próbników: *livenessProbe* i *readinessProbe*. Pierwszy określa, czy kontener działa, jeżeli nie to powinien być zrestartowany. Drugi określa czy kontener jest gotowy do obsługi zapytań, kontener jest wyrejestrowywany z *Service* na czas nieprzechodzenia *readinessProbe*.

ReplicaSet *ReplicaSet* jest następcą *ReplicaControllera*, czyli imperatywnym kontrolerem dbającym o działanie określonej liczby *Podów* w klastrze.

Jest to bardzo prosty kontroler i nie powinien być używany bezpośrednio.

Deployment *Deployment* pozwala na deklaratywne aktualizacje *Podów* i *ReplicaSetów*. Korzystanie z ww. bezpośrednio nie jest zalecane.

Zmiany *Deploymentów* wprowadzane są przez tak zwane *rollouty*. Każdy ma swój status i może zostać wstrzymany lub przerwany. *Rollouty* mogą zostać aktywowane automatycznie przez zmianę specyfikacji *Pod_a* przez *.spec.template_*.

Rewizje *_Deployment_u* są zmieniane tylko w momencie *_rollout_u*. Operacja operacja skalowania nie uruchamia *_rollout_u*, a więc nie zmienia rewizji.

Podstawowe przypadki użycia *Deployment* to:

- uruchamianie *ReplicaSet*ów w tle przez *.spec.replicas*,
- deklarowanie nowego stanu *Pod*ów zmieniając *.spec.template*,
- cofanie zmian do poprzednich rewizji *_Deployment_*u (poprzednie wersje *Pod*ów) komendą *kubectl rollout undo*,
- skalowanie *_Deployment_*u w celu obsługi większego obciążenia przykładową komendą *kubectl autoscale deployment nginx-deployment --min=10 --max=15 --cpu-percent=80*,
- wstrzymywanie *Deployment* w celu wprowadzenia poprawek komendą *kubectl rollout pause deployment/nginx-deployment*,
- czyszczenie historii *ReplicaSet*ów przez ograniczanie liczby wpisów w *.spec.revisionHistoryLimit*,

Przykładowy *Deployment* tworzący 3 repliki serwera *nginx*:

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5   labels:
6     app: nginx
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: nginx
12   template:
13     metadata:
14       labels:
15         app: nginx
16     spec:
17       containers:
18         - name: nginx
19           image: nginx:1.7.9
20           ports:
21             - containerPort: 80

```

Pole *.spec.selector* definiuje w jaki sposób *Deployment* ma znaleźć *Pody*, którymi ma zarządzać. Selektor powinien zgadzać się ze zdefiniowanym szablonem.

StatefulSet *StatefulSet* jest kontrolerem podobnym do *_Deployment_*u, ale umożliwiającym zachowanie stanu *Pod*ów.

W przeciwieństwie do *_Deployment_*u *StatefulSet* nadaje każdemu uruchomionemu *_Pod_*owi stały unikalny identyfikator, który zostają zachowane mimo restartów i przenoszenia *Pod*ów. Identyfikatory można zastosować między innymi do:

- trwałych i unikalnych identyfikatorów wewnątrz sieci,
- trwałych zasobów dyskowych,
- sekwencyjne uruchamianie i skalowanie aplikacji,
- sekwencyjne zakańczanie i usuwanie aplikacji,
- sekwencyjne, zautomatyzowane aktualizacje aplikacji,

DaemonSet *DaemonSet* jest kontrolerem upewniającym się, że przynajmniej jeden *Pod* działa na każdym lub wybranych węzłach klastra.

Do jego typowych zastosowań należy implementacja narzędzi wymagających agenta na każdym z węzłów:

- rozproszone systemy dyskowe, np. *glusterd*, *ceph*,
- zbieracze logów, np. *fluentd*, *logstash*,
- monitorowanie węzłów, np. *Prometheus Node Exporter*, *collectd*,

Job i CronJob *Job* pozwala na jednorazowe uruchomienie *Podów*, które wykonują akcję i się kończą. Istnieją 3 tryby wykonania: niezrównoległony, równoległy i równoległy z zewnętrzną kolejką zadań.

Domyślnie przy niepowodzeniu uruchamiane są kolejne *Pody* aż zostanie uzyskana odpowiednia liczba sukcesów.

CronJob pozwala na tworzenie *Jobów* jednorazowo o określonym czasie lub je powtarzać zgodnie ze specyfikacją *cron*.

3.5 *Kubernetes Dashboard*

Kubernetes Dashboard jest wbudowanym interfejsem graficznym klastra *Kubernetes*. Umożliwia monitorowanie i zarządzanie klastrem w ramach funkcjonalności samego *Kubernetesa*.

3.6 *Kubernetes Incubator*

Kubernetes Incubator gromadzi projekty rozszerzające *Kubernetes*, ale nie będące częścią oficjalnej dystrybucji. Został stworzony, aby opanować bałagan w głównym repozytorium oraz ujednolicić proces tworzenia rozszerzeń.

Aby dołączyć do inkubatora projekt musi spełnić szereg wymagań oraz nie może spędzić w inkubatorze więcej niż 18 miesięcy. Dostępne opcje opuszczenia inkubatora to:

- awansować do rangi oficjalnego projektu *Kubernetesa*,
- połączyć się z istniejącym oficjalnym projektem,

- po 12 miesiącach przejść w stan spoczynku, a po kolejnych 6 miesiącach zostać przeniesiony do *kubernetes-incubator-retired*

3.7 Administracja klastrem z linii komend

kubeadm

kubeadm jest narzędziem pozwalającym na niskopoziomowe zarządzanie klastrem *Kubernetesa*. Stąd trendem jest bazowanie na kubeadm przy tworzeniu narzędzi z wyższym poziomem abstrakcji.

- Install with kubadm

Kubespray

kubespray jest zbiorem skryptów Ansibla konfigurujących klastry na różnych systemach operacyjnych i w różnych konfiguracjach. W tym jest w stanie skonfigurować klastery bare metal bez żadnych zewnętrznych zależności.

Projekt na dzień dzisiejszy znajduje się w inkubatorze i jest aktywnie rozwijany.

OpenShift Ansible

Konfiguracja OpenShift Origin realizowana jest zestawem skryptów Ansible'owych rozwijanych jako projekt openshift-ansible.

Canonical distribution of Kubernetes

Jest to prosta w instalacji dystrybucja *Kubernetesa*. Niestety wymaga infrastruktury chmurowej do uruchomienia klastra składającego się z więcej niż jednego węzła.

Opcja *bare metal*, która by mnie interesowała nadal wymaga działającego środowiska Metal as a Service.

W związku z powyższym nie będę dalej zajmował się tym narzędziem.

Materiały źródłowe:

- pakiet (*Charm*) w oficjalnym repozytorium *Juju*
- materiał szkoleniowy dot. uruchamiania *Kubernetesa*
- opis instalacji lokalnego klastra

Bootkube i Typhoon

Bootkube jest narzędziem napisanym w języku *Go* pozwalającym skonfigurować *Kubernetes* na własnych maszynach.

W instalacji *bare metal* proponowane jest wykorzystanie *Terraform* i *Typhoon* do realizacji automatycznej konfiguracji klastra w trakcie procesu uruchamiania węzłów *CoreOS*.

Domyślnie ww. narzędzia konfigurują instalację *CoreOS* na dysku, a następnie restartują maszynę.

W wyniku przeoczenia wzmianki (przypis na jednej z podstron dokumentacji) o możliwości uruchomienia w trybie bezdyskowym całkowicie odrzuciłem to narzędzie. W końcowych etapach pisania pracy znalazłem ww. wpis i zdecydowałem się zawrzeć o nim informację.

Eksperymentalne i deprekowane rozwiązania

- *Fedora via Ansible* deprekowane na rzecz *kubespray*
- *Rancher Kubernetes Installer* jest eksperymentalnym rozwiązaniem wykorzystywanym w *Rancher 2.0*,

kubespray-cli Jest to narzędzie ułatwiające korzystanie z *kubespray*. Według użytkowników oficjalnego *Slacka kubespray kubespray-cli* jest deprekowane i powinno się korzystać z czystego *kubespray*.

3.8 Administracja klastrem za pomocą narzędzi graficznych

Rancher

Rancher jest platformą zarządzania kontenerami umożliwiającą między innymi zarządzanie klastrem *Kubernetesa*. Od wersji 2.0 twórcy skupiają się wyłącznie na zarządzaniu *Kubernetesem* porzucając wsparcie innych rozwiązań.

OpenShift by Red Hat

OpenShift jest komercyjną usługą typu *PaaS* (Platform as a Service), od wersji 3 skupia się na zarządzaniu klastrem *Kubernetesa*.

Rdzeniem projektu jest open sourcowy OpenShift Origin konfigurowany przez OpenShift Ansible.

Materiały źródłowe:

- dyskusja o wykorzystaniu *OpenShift Origin* i *Kubernetes*
- opis różnic między *OpenShift Origin* i *Kubernetes*
- materiał wideo przedstawiający interfejs OpenShift (po hebrajsku)

DC/OS

Datacenter Operating System jest częścią Mesosphere i Mesosa. Niedawno został rozszerzony o *Kubernetesa* jako alternatywny (w stosunku do *Marathon*) system orkiestracji kontenerami.

3.9 Lista materiałów dodatkowych

Ze względu na obszerność tematu zdecydowałem przedstawić oddzielną listę materiałów dodatkowych:

- blog Julii Evans o *Kubernetes*,
- dokument o uruchamianiu *Kubernetes* od podstaw,
- materiał wideo o skalowaniu *Kubernetes*,

Rozdział 4

Systemy bezdyskowe

Maszyny bezdyskowe jak nazwa wskazuje nie posiadają lokalnego medium trwałego przechowywania informacji. W związku z tym wszystkie informacje są przechowywane w pamięci RAM komputera i są tracone w momencie restartu maszyny.

System operacyjny musi wspierać uruchamianie w takim środowisku. Wiele systemów nie wspiera tego trybu operacyjnego zakładając obecność dysku twardego w maszynie.

W niektórych przypadkach mimo braku domyślnego wsparcia istnieje możliwość przygotowania własnego obrazu systemu operacyjnego wspierającego ten tryb pracy:

- Fedora Atomic Host.

Potencjalnymi rozwiązaniami problemu przechowywania stanu maszyn bezdyskowych mogą być:

- przydziały NFS,
- replikacja ZFS,
- przechowywanie całego stanu w cloud-init.

4.1 Proces uruchamiania maszyny bezdyskowej

Na uruchamianie maszyn bezdyskowych w protokole PXE składają się 3 podstawowe elementy:

1. serwer DHCP, np. isc-dhcp-server lub dnsmasq,
2. firmware wspierające PXE, np. iPXE,
3. serwer plików (np. TFTP, HTTP, NFS) i/lub sieciowej pamięci masowej (np. iSCSI).

Pełną lokalną konfigurację bazowaną na Dockerze przechowuję w moim repozytorium ipxe-boot.

Rozdział 5

Przegląd systemów operacyjnych

Wszystkie moduły *Kubernetes* są uruchamiane w kontenerach, więc dwoma podstawowymi wymaganiami systemu operacyjnego są:

- możliwość instalacji i uruchomienia Dockera,
- wsparcie wybranego narzędzia konfigurującego system do działania w klastrze *Kubernetes*,

Dodatkowe wymagania związane z opisywanym w tej pracy przypadkiem użycia:

- zdalny dostęp SSH lub możliwość konfiguracji automatycznego dołączenia do klastra *Kubernetesa*,
- wsparcie dla środowiska bezdyskowego,
- możliwość bootu PXE.

Podstawowe wyznaczniki:

- sposób konfiguracji maszyny,
- rozmiar minimalnego działającego systemu spełniającego wszystkie wymagania,
- aktualne wersje oprogramowania.

5.1 Konfigurator cloud-init

Ze względu na obszerność i niejednoznaczność tematu cloud-init zdecydowałem się wyjaśnić wszelkie wątpliwości z nim związane.

cloud-init jest standardem oraz implementacją konfiguratora kompatybilnego z wieloma systemami operacyjnymi przeznaczonymi do działania w chmurze.

Standard polega na dostarczeniu pliku konfiguracyjnego w formacie YAML w trakcie lub tuż po inicjalizacji systemu operacyjnego.

Główną zaletą cloud-init jest tworzenie automatycznej i jednolitej konfiguracji bazowych systemów operacyjnych w środowiskach chmurowych, czyli częstego podnoszenia nowych maszyn.

Dostępne implementacje

cloud-init Referencyjny *cloud-init* zaimplementowany jest w Pythonie, co częściowo tłumaczy duży rozmiar obrazów przeznaczonych dla chmury. Po najmniejszych obrazach *Pythona* dla *Dockera* (`python:alpine` - 89MB i `python2:alpine` - 72 MB) wnioskuję, że nie istnieje mniejsza dystrybucja *Pythona*.

```
1 docker pull python:2-alpine > /dev/null
2 docker pull python:alpine > /dev/null
3 docker images | grep alpine
```

Dodatkowe materiały:

- Wywiad z developerem cloud-init

coreos-cloudinit *coreos-cloudinit* jest częściową implementacją standardu w języku Go udostępnioną przez twórców *CoreOS*. Rok temu przestał być rozwijany i wychodzi z użytku.

RancherOS + coreos-cloudinit *Rancher cloud-init* jest przejętym *coreos-cloudinit* przez zespół *RancherOS*.

clr-cloud-init *clr-cloud-init* jest wewnętrzną implementacją standardu dla systemu *ClearLinux*. Powstała z chęci optymalizacji standardu pod *ClearLinux* oraz pozbycia się zależności referencyjnej implementacji od *Pythona*.

5.2 CoreOS

CoreOS jest pierwszą dystrybucją *Linuxa* przeznaczoną do zarządzania kontenerami. Zawiera dużo narzędzi dedykowanych klastrowaniu i obsłudze kontenerów, w związku z tym zajmuje 342 MB.

Czysta instalacja zajmuje około 600 MB pamięci RAM i posiada najnowsze wersje *Dockera* i *OverlayFS*.

30 stycznia 2018 roku został wykupiony przez Red Hat.

Konfiguracja

Konfiguracja obsługiwana jest przez Container Linux Config transpilowany do Ignition. Transpiler konwertuje ogólną konfigurację na przygotowaną pod konkretne chmury (AWS, GCE, Azure itp.). Minusem jest brak dystrybucji transpilatora pod *FreeBSD*.

Poprzednikiem Ignition jest *coreos-cloudinit*.

5.3 RancherOS

RancherOS jest systemem operacyjnym, w którym tradycyjny system inicjalizacji został zastąpiony trzema poziomami Dockera:

- *bootstrap_docker* - działający w initramie, czyli przygotowuje system,
- *system-docker* - zastępuje tradycyjny init, zarządza wszystkimi programami systemowymi,
- *docker* - standardowy *Docker*, interakcja z nim nie może uszkodzić działającego systemu.

Jego głównymi zaletami są mały rozmiar plików startowych (45 MB) oraz prostota konfiguracji.

Czysta instalacja zajmuje około 700 MB pamięci RAM. Niestety nie jest często aktualizowany i posiada stare wersje zarówno Dockera (17.06 sprzed pół roku) jak i *overlay* (zamiast *overlay2*).

W związku z bugiem w systemie RancherOS nie zawsze czyta *cloud-config*, więc odrzucam ten system operacyjny w dalszych rozważaniach.

Konfiguracja

RancherOS jest konfigurowany przez własną wersję *coreos-cloudinit*.

Znaczną przewagą wobec oryginału jest możliwość sekwencyjnego uruchamiania dowolnej liczby plików konfiguracyjnych.

Minimalna konfiguracja pozwalająca na zalogowanie:

```
1 #cloud-config
2 ssh_authorized_keys:
3   - ssh-rsa AAAAB3N...
```

Generuję ją poniższym skrypcem na podstawie komendy *ssh-add -L*:

```
1 #!/bin/sh
2
3 cat << EOF > ssh.yml
4 #cloud-config
```



```
5 ssh_authorized_keys:
6 $(ssh-add -L | sed 's/^/ - /g')
7 EOF
```

Przydatne jest wyświetlenie kompletnej konfiguracji komendą *ros config export -full*.

5.4 *Project Atomic*

Project Atomic jest grupą podobnie skonfigurowanych systemów operacyjnych dedykowaną środowiskom cloud i kontenerom.

Dystrybucje *Project Atomic* nazywają się *Atomic Host*. Dostępne są ich następujące warianty:

- Red Hat Atomic Host,
- CentOS Atomic Host,
- Fedora Atomic Host.

Żadna z dystrybucji domyślnie nie wspiera rozruchu bezdyskowego, więc nie zgłębiam dalej tematu.

Atomic Host są konfigurowane oficjalną implementacją *cloud-inita*.

5.5 Alpine Linux

Alpine Linux jest minimalną dystrybucją Linuxa bazowaną na musl-libc i busybox.

Wygląda bardzo obiecująco w kontekście moich zastosowań, ale ze względu na błąd w procesie inicjalizacji systemu aktualnie nie ma możliwości jego uruchomienia w trybie bezdyskowym.

Alpine Linux może być skonfigurowany przez Alpine Backup lub Alpine Configuration Framework.

5.6 ClearLinux

ClearLinux jest dystrybucją *Linuxa* wysoko zoptymalizowaną pod procesory Intel.

Poza intensywną optymalizacją ciekawy w tej dystrybucji jest koncept *bundle* zamiast standardowych pakietów systemowych. Żaden z bundli nie może zostać zaktualizowany oddzielnie, w zamian cały system operacyjny

jest aktualizowany na raz ze wszystkimi bundlami. Znacznie ułatwia to zarządzanie wersjami oprogramowania i stanem poszczególnych węzłów sieci komputerowej.

Czysta instalacja z Dockerem i serwerem SSH również zajmuje 700 MB pamięci RAM więc nie odbiega od innych dystrybucji.

Ogromnym minusem jest trudna w nawigowaniu dokumentacja systemu operacyjnego.

Materiały źródłowe:

- 6 key points about Intel's hot new Linux distro

5.7 Wnioski

Głównymi czynnikami odróżniającymi poszczególne systemy operacyjne są częstotliwość aktualizacji oprogramowania oraz wsparcie narzędzi. Rozbieżność reszty parametrów jest pomijalnie mała.

Najczęściej aktualizowanym z powyższych systemów jest CoreOS, więc na nim skupię się w dalszej części pracy.

Rozdział 6

Praktyczne rozeznanie w narzędziach administracji klastrem *Kubernetesa*

Najpopularniejszym rozwiązaniem konfiguracji klastra *Kubernetes* jest *kops*, ale jak większość rozwiązań zakłada uruchomienie w środowiskach chmurowych, *PaaS* lub *IaaS*. W związku z tym nie ma żadnego zastosowania w tej pracy inżynierskiej.

6.1 kubespray-cli

Z powodu błędu logiki narzędzie nie radzi sobie z brakiem *Pythona* na domyślnej dystrybucji *CoreOSa*, mimo że sam *kubespray* radzi sobie z nim świetnie.

Do uruchomienia na tym systemie potrzebne jest ręczne wywołanie roli *bootstrap-os* z *kubespray* zanim przystąpi się do właściwego deployment'u. Skrypt uruchamiający:

```
1 #!/bin/sh
2 set -e
3
4 # pip2 install ansible kubespray
5 get_coreos_nodes() {
6     for node in $@
7     do
8         echo -n node1[
9         echo -n ansible_host=${node},
10        echo -n bootstrap_os=coreos,
11        echo -n ansible_user=core,
12        echo -n ansible_default_ipv4.address=${node}
```

```

13     echo ]
14 done
15 }
16
17 NODES=$(get_coreos_nodes 192.168.56.{10,12,13})
18 echo NODES=${NODES[@]}
19 kubespray prepare -y --nodes ${NODES[@]}
20
21 cat << EOF > ~/.kubespray/bootstrap-os.yml
22 - hosts: all
23   become: yes
24   gather_facts: False
25   roles:
26   - bootstrap-os
27 EOF
28
29 (
30     cd ~/.kubespray;
31     ansible-playbook -i inventory/inventory.cfg bootstrap-os.
32         ↪ yml
33 )
34 kubespray deploy -y --coreos

```

Napotkane problemy

Narzędzie kończy się błędem na kroku czekania na uruchomienie *etcd*, ponieważ oczekuje połączenia na NATowym interfejsie z adresem *10.0.3.15* zamiast host network z adresem *192.168.56.10*, stąd ręczne podawanie *ansible_default_ipv4.address*.

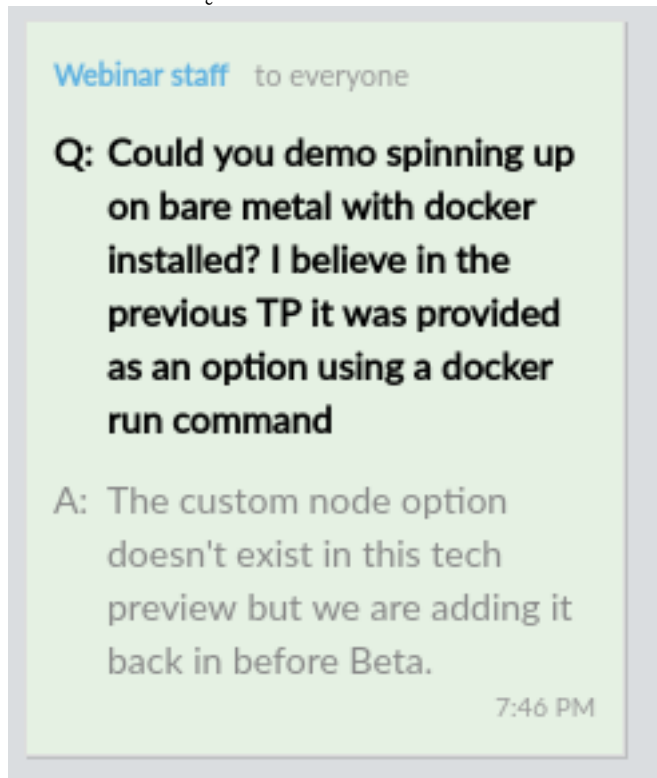
Wnioski

W trakcie testowania okazało się, że *kubespray-cli* nie jest aktywnie rozwijane i stało się niekompatybilne z samym projektem *Kubespray*. W związku z tym uznaję *kubespray-cli* za nie mające zastosowania w tej pracy inżynierskiej.

6.2 Rancher 2.0

Jest to wygodne narzędzie do uruchamiania i monitorowania klastra *Kubernetesa*, ale wymaga interakcji użytkownika. Wersja 2.0 (obecnie w fazie alpha) oferuje lepszą integrację z *Kubernetesem* całkowicie porzucając inne platformy.

W trakcie pisania pracy (24 stycznia 2018) pojawiło się drugie Tech Preview. W stosunku do pierwszego Tech Preview aplikacja została mocno przebudowana i nie wspiera jeszcze konfiguracji *bare metal*, więc jestem zmuszony odrzucić to rozwiązanie.



Testowanie tech preview 1 (v2.0.0-alpha10)

```
1 #rancher_version=latest
2 #rancher_version=preview
3 rancher_version=v2.0.0-alpha10
4 docker run --rm --name rancher -d -p 8080:8080 rancher/
  ↪ server:${rancher_version}
```

Najpierw należy zalogować się do panelu administracyjnego Ranchera i przeprowadzić podstawową konfigurację (adres Ranchera + uzyskanie komendy).

Następnie w celu dodania węzła do klastra wystarczy wywołać jedną komendę udostępnioną w panelu administracyjnym Ranchera na docelowym węźle, jej domyślny format to:

```
1 wersja_agenta=v1.2.9
2 ip_ranchera=192.168.56.1
```

```

3 skrypt=B52944BEFAA613F0CE90:1514678400000:
  ↪ E2yB6KfxzSix4YHti39BTw5RbKw
4
5 sudo docker run --rm --privileged \
6   -v /var/run/docker.sock:/var/run/docker.sock \
7   -v /var/lib/rancher:/var/lib/rancher \
8   rancher/agent:${wersja_agenta} \
9   http://${ip_ranchera}:8080/v1/scripts/${skrypt}

```

W ciągu 2 godzin przeglądu nie udało mi się zautomatyzować procesu uzyskiwania ww. komendy.

Następnie w *cloud-config RancherOSa* możemy dodać ww. komendę w formie:

```

1 #cloud-config
2 runcmd:
3 - docker run --rm --privileged -v /var/run/docker.sock:/var
  ↪ /run/docker.sock -v /var/lib/rancher:/var/lib/rancher
  ↪ rancher/agent:v1.2.9 http://192.168.56.1:8080/v1/
  ↪ scripts/...

```

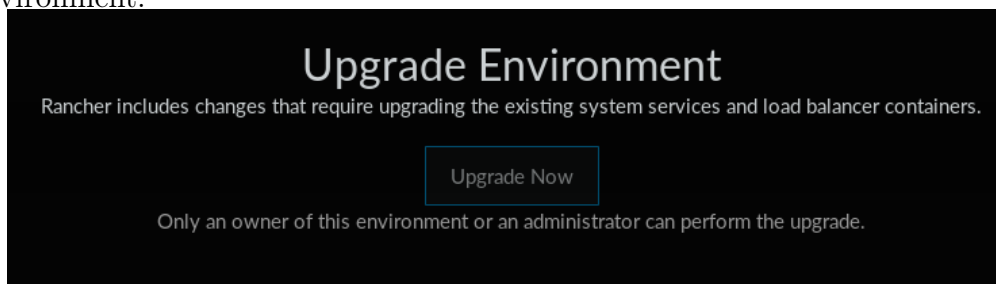
Od wersji 2.0 umożliwia połączenie się z istniejącym klastrem:

```

1 kubectl apply -f http://192.168.56.1:8080/v3/scripts/303
  ↪ F60E1A5E186F53F3F:1514678400000:
  ↪ wstQFdHpOgHqKahoYdmsCXEWMW4.yaml

```

Napotkane błędy W wersji *v2.0.0-alpha10* losowo pojawia się błąd Upgrade Environment.



Wnioski

Rancher na chwilę obecną (styczeń 2018 roku) jest bardzo wygodnym, ale również niestabilnym rozwiązaniem.

Ze względu na brak stabilności odrzucam Ranchera jako rozwiązanie problemu uruchamiania klastra *Kubernetesa*.

6.3 OpenShift Origin

Według dokumentacji są dwie metody uruchamiania serwera, w *Dockerze* i bezpośrednio na systemie *Linux*.

```
1 # https://docs.openshift.org/latest/getting\_started/  
  ↪ administrators.html#installation-methods  
2 docker run -d --name "origin" \  
3   --privileged --pid=host --net=host \  
4   -v /:/rootfs:ro \  
5   -v /var/run:/var/run:rw \  
6   -v /sys:/sys \  
7   -v /sys/fs/cgroup:/sys/fs/cgroup:rw \  
8   -v /var/lib/docker:/var/lib/docker:rw \  
9   -v /var/lib/origin/openshift.local.volumes:/var/lib/  
  ↪ origin/openshift.local.volumes:rslave \  
10  openshift/origin start --public-master
```

Dodałem opcję `-public-master` aby uruchomić konsolę webową

Korzystanie ze sterownika systemd zamiast domyślnego cgroupfs

Większość dystrybucji *Linuxa* (np. Arch, CoreOS, Fedora, Debian) domyślnie nie konfiguruje sterownika cgroup Dockera i korzysta z domyślnego *cgroupfs*.

Typ sterownika cgroup można wyświetlić komendą `docker info`:

```
1 $ docker info | grep -i cgroup  
2 Cgroup Driver: systemd
```

OpenShift natomiast konfiguruje *Kubernetesa* do korzystania z *cgroup* przez *systemd*. Kubelet przy starcie weryfikuje zgodność silników cgroup, co skutkuje niekompatybilnością z domyślną konfiguracją Dockera, czyli poniższym błędem:

```
1 F0120 19:18:58.708005 25376 node.go:269] failed to run  
  ↪ Kubelet: failed to create kubelet: misconfiguration:  
  ↪ kubelet cgroup driver: "systemd" is different from  
  ↪ docker cgroup driver: "cgroupfs"
```

Problem można rozwiązać dopisując `-exec-opt native.cgroupdriver=systemd` do linii komend `dockerd` (zwykle w pliku `docker.service`). Dla przykładu w *Arch Linuxie* zmiana wygląda następująco:

```
1 $ cp /usr/lib/systemd/system/docker.service /etc/systemd/  
  ↪ system/docker.service  
2 $ vim /etc/systemd/system/docker.service  
3 $ diff /usr/lib/systemd/system/docker.service /etc/systemd/  
  ↪ system/docker.service  
4 13c13
```

```

5 < ExecStart=/usr/bin/dockerd -H fd://
6 ---
7 > ExecStart=/usr/bin/dockerd -H fd:// --exec-opt native.
    ↪ cgroupdriver=systemd

```

Próba uruchomienia serwera na Arch Linux

Po wystartowaniu serwera zgodnie z dokumentacją OpenShift Origin i naprawieniu błędu z konfiguracją cgroup przeszedłem do kolejnego kroku Try It Out:

1. Uruchomienie shella na serwerze:

```
1 $ docker exec -it origin bash
```

2. Logowanie jako testowy użytkownik:

```

1 $ oc login
2 Username: test
3 Password: test

```

3. Stworzenie nowego projektu:

```
1 $ oc new-project test
```

4. Pobranie aplikacji z Docker Hub:

```

1 $ oc tag --source=docker openshift/deployment-example:v1
    ↪ deployment-example:latest

```

5. Wystartowanie aplikacji:

```
1 $ oc new-app deployment-example:latest
```

6. Oczekanie aż aplikacja się uruchomi:

```

1 $ watch -n 5 oc status
2 In project test on server https://192.168.0.87:8443
3
4 svc/deployment-example - 172.30.52.184:8080
5   dc/deployment-example deploys istag/deployment-example:
    ↪ latest
6   deployment #1 failed 1 minute ago: config change

```

Niestety nie udało się przejść kroku 5, więc próba uruchomienia OpenShift Origin na Arch Linux zakończyła się niepowodzeniem.

Próba uruchomienia serwera na *Fedora Atomic Host* w *Virtual-Boksie*

Maszynę z najnowszym *Fedora Atomic Host* uruchomiłem za pomocą poniższego *Vagrantfile*:

```
1 # -*- mode: ruby -*-
2 # vi: set ft=ruby :
3
4 Vagrant.configure("2") do |config|
5   config.vm.box = "fedora/27-atomic-host"
6   config.vm.box_check_update = false
7   config.vm.network "forwarded_port", guest: 8443, host:
8     ↳ 18443, host_ip: "127.0.0.1"
9   config.vm.network "forwarded_port", guest: 8080, host:
10     ↳ 18080, host_ip: "127.0.0.1"
11   config.vm.provider "virtualbox" do |vb|
12     vb.gui = false
13     vb.memory = "8192"
14   end
15   config.vm.provision "shell", inline: <<-SHELL
16   SHELL
17 end
```

```
1 $ vagrant up
2 $ vagrant ssh
3 $ sudo docker run -d --name "origin" \
4   --privileged --pid=host --net=host \
5   -v /:/rootfs:ro \
6   -v /var/run:/var/run:rw \
7   -v /sys:/sys \
8   -v /sys/fs/cgroup:/sys/fs/cgroup:rw \
9   -v /var/lib/docker:/var/lib/docker:rw \
10  -v /var/lib/origin/openshift.local.volumes:/var/lib/
11     ↳ origin/openshift.local.volumes:rslave \
12  openshift/origin start
```

Kroki 1-5 były analogiczne do uruchamiania na *Arch Linux*, następnie:

6. Oczekanie aż aplikacja się uruchomi i weryfikacja działania:

```
1 $ watch -n 5 oc status
2 In project test on server https://10.0.2.15:8443
3
4 svc/deployment-example - 172.30.221.105:8080
5   dc/deployment-example deploys istag/deployment-example:
6     ↳ latest
7   deployment #1 deployed 3 seconds ago - 1 pod
8 $ curl http://172.30.221.105:8080 | grep v1
9 <div class="box"><h1>v1</h1><h2></h2></div>
```

7. Aktualizacja, przebudowanie i weryfikacja działania aplikacji:

```
1 $ oc tag --source=docker openshift/deployment-example:v2
   ↳ deployment-example:latest
2 Tag deployment-example:latest set to openshift/deployment-
   ↳ example:v2.
3 $ watch -n 5 oc status
4 In project test on server https://10.0.2.15:8443
5
6 svc/deployment-example - 172.30.221.105:8080
7   dc/deployment-example deploys istag/deployment-example:
   ↳ latest
8     deployment #2 running for 8 seconds - 1 pod
9     deployment #1 deployed 8 minutes ago - 1 pod
10 $ curl -s http://172.30.221.105:8080 | grep v2
11 <div class="box"><h1>v2</h1><h2></h2></div>
```

8. Nie udało się uzyskać dostępu do panelu administracyjnego OpenShift:

```
1 $ curl -k 'https://localhost:8443/console/'
2 missing service (service "webconsole" not found)
3 missing route (service "webconsole" not found)
```

W internecie nie znalazłem żadnych informacji na temat tego błędu. Próbowałem również uzyskać pomoc na kanale *#openshift* na *irc.freenode.net*, ale bez skutku.

Wnioski

Panel administracyjny klastra *OpenShift Origin* jest jedyną znaczącą przewagą nad *Kubespriem*. Reszta zarządzania klastrem odbywa się również za pomocą repozytorium skryptów Ansibla (w tym dodawanie kolejnych węzłów klastra).

Z powodu braku dostępu do ww. panelu próbę uruchomienia *OpenShift Origin* uznaję za nieudaną i odrzucam to narzędzie.

6.4 kubespriem

Cały kod znajduje się w moim repozytorium *kubernetes-cluster*.

Kubernetes Dashboard

Dostęp do Dashboardu najprościej można uzyskać poprzez:

1. nadanie wszystkich uprawnień roli *kubernetes-dashboard*,
2. Wejście pod adres *http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/login* ,
3. Kliknięcie skip.

Materiały źródłowe:

- <https://github.com/kubernetes/dashboard/wiki/Access-control>
- <https://github.com/kubernetes-incubator/kubespray/blob/master/docs/getting-started.md#accessing-kubernetes-dashboard>

Napotkane błędy

Błąd przy ustawieniu *loadbalancer_apiserver.address* na *0.0.0.0*:

```
1 TASK [kubernetes-apps/cluster_roles : Apply workaround to
    ↪ allow all nodes with cert 0=system:nodes to register]
    ↪ *****
2 Wednesday 17 January 2018  22:22:59 +0100 (0:00:00.626)
    ↪ 0:08:31.946 *****
3 fatal: [node2]: FAILED! => {"changed": false, "msg": "error
    ↪ running kubect1 (/opt/bin/kubect1 apply --force --
    ↪ filename=/etc/kubernetes/node-crb.yml) command (rc=1)
    ↪ : Unable to connect to the server: http: server gave
    ↪ HTTP response to HTTPS client\n"}
4 fatal: [node1]: FAILED! => {"changed": false, "msg": "error
    ↪ running kubect1 (/opt/bin/kubect1 apply --force --
    ↪ filename=/etc/kubernetes/node-crb.yml) command (rc=1)
    ↪ : Unable to connect to the server: http: server gave
    ↪ HTTP response to HTTPS client\n"}
```

6.5 Wnioski

Na moment pisania tej pracy *Kubespray* jest jedynym aktywnie rozwijanym i działającym rozwiązaniem uruchamiania klastra *Kubernetesa*.

Rozdział 7

Uruchamianie *Kubernetesa* w laboratorium 225

7.1 Przygotowanie węzłów *CoreOS*

Na wstępie przygotowałem *coreos.ipxe* i *coreos.ign* do rozruchu i bezhasłowego dostępu.

Po pierwsze stworzyłem Container Linux Config (plik *coreos.yml*) zawierający:

1. Tworzenie użytkownika *nazarewk*,
2. Nadanie mu praw do *sudo* i *dockera* (grupy *sudo* i *docker*),
3. Dodanie dwóch kluczy: wewnętrznego uczelnianego i mojego używanego na codzień w celu zdalnego dostępu.

```
1 passwd:
2   users:
3     - name: nazarewk
4       groups: [sudo, docker]
5       ssh_authorized_keys:
6         - ssh-rsa ... nazarewk
7         - ssh-rsa ... nazarewk@ldap.iem.pw.edu.pl
```

Następnie skompilowałem go do formatu Ignition narzędziem *ct*, skryptem *bin/render-coreos* z wykazu.

Przygotowałem skrypt *IPXE* do uruchamiania CoreOS *zetis/WWW/boot/coreos.ipxe*.

Umieściłem skrypt w */home/stud/nazarewk/WWW/boot* i wskazałem go maszynom, które będą węzłami:

```
1 sudo lab 's4 s5 s6 s8 s9' boot http://vol/~nazarewk/boot/
   ↪ coreos.ipxe
```

7.2 Przeszkody związane z uruchamianiem skryptów na uczelnianym Ubuntu

Brak virtualenv'a

Moje skrypty nie przewidywały braku *virtualenv*, więc musiałem ręcznie zainstalować go komendą `apt-get install virtualenv`. Dodalem ten krok do skryptu *setup-packages*.

Klonowanie repozytorium bez logowania

W celu umożliwienia anonimowego klonowania repozytorium z Githuba, zmieniłem protokół z *git* na *https*:

```
1 git clone https://github.com/nazarewk/kubernetes-cluster.  
  ↪ git
```

Problem pojawił się również dla submodułów gita (*.gitmodules*).

Atrybut wykonywalności skryptów

W konfiguracji uczelnianej git nie ustawia domyślnie atrybutu wykonalności dla plików wykonywalnych i zdejmuje go przy aktualizacji pliku. Problem rozwiązałem dodaniem komendy `chmod +x bin/*` do skryptu *pull*.

Konfiguracja dostępu do maszyn bez hasła

Poza konfiguracją *CoreOS* wypełniłem konfigurację SSH do bezhasłowego dostępu. W pliku `~/.ssh/config` umieściłem:

```
1 Host s?  
2   User admin  
3  
4 IdentityFile ~/.ssh/id_rsa  
5 IdentitiesOnly yes  
6  
7 Host s?  
8   StrictHostKeyChecking no  
9   UserKnownHostsFile /dev/null
```

Problemy z siecią

W trakcie pierwszego uruchamiania występowały problemy z siecią uczelnianą, więc rozszerzyłem plik *ansible.cfg* o ponawianie prób wywoływania komend dodając wpis *retries=5* do sekcji *[ssh_connection]*.

Limit 3 serwerów DNS

Napotkałem limit 3 serwerów DNS:

```
1 TASK [docker : check system nameservers]
  ↳ *****
2 Friday 26 January 2018 14:47:09 +0100 (0:00:01.429)
  ↳ 0:04:26.879 *****
3 ok: [node3] => {"changed": false, "cmd": "grep \"^
  ↳ nameserver\" /etc/resolv.conf | sed 's/^nameserver\\s
  ↳ *//'", "delta": "0:00:00.004652", "end": "2018-01-26
  ↳ 13:47:11.659298", "rc": 0, "start": "2018-01-26
  ↳ 13:47:11.654646", "stderr": "", "stderr_lines": [], "
  ↳ stdout": "172.29.146.3\n1
4 72.29.146.6\n10.146.146.3\n10.146.146.6", "stdout_lines":
  ↳ ["172.29.146.3", "172.29.146.6", "10.146.146.3",
  ↳ "10.146.146.6"]}
5 ...
6 TASK [docker : add system nameservers to docker options]
  ↳ *****
7 Friday 26 January 2018 14:47:13 +0100 (0:00:01.729)
  ↳ 0:04:30.460 *****
8 ok: [node3] => {"ansible_facts": {"docker_dns_servers":
  ↳ ["10.233.0.3", "172.29.146.3", "172.29.146.6",
  ↳ "10.146.146.3", "10.146.146.6"]}, "changed": false}
9 ...
10 TASK [docker : check number of nameservers]
  ↳ *****
11 Friday 26 January 2018 14:47:15 +0100 (0:00:01.016)
  ↳ 0:04:32.563 *****
12 fatal: [node3]: FAILED! => {"changed": false, "msg": "Too
  ↳ many nameservers. You can relax this check by set
  ↳ docker_dns_servers_strict=no and we will only use the
  ↳ first 3."}
```

Okazało się, że maszyna *s8* była podłączona również na drugim interfejsie sieciowym, w związku z tym miała zbyt dużo wpisów serwerów DNS.

Rozwiązałem problem ręcznie logując się na maszynę i wyłączając drugi interfejs sieciowy komendą `ip 1 set eno1 down`.

7.3 Pierwszy dzień - uruchamianie skryptów z maszyny *s6*

Większość przeszkód opisałem w powyższym rozdziale, więc w tym skupię się tylko na problemach związanych z pierwszą próbą uruchomienia skryptów na maszynie *s6*.

Najpierw próbowałem uruchomić skrypty na maszynach: *s2*, *s4* i *s5*

```
1 cd ~/kubernetes/kubernetes-cluster
2 bin/setup-cluster-full 10.146.255.{2,4,5}
```

Po uruchomieniu okazało się, że maszyna *s2* posiada tylko połowę RAMu (4GB) i nie mieszczą się na niej obrazy Dockera konieczne do uruchomienia klastra.

Kolejną próbą było uruchomienie na maszynach *s4*, *s5*, *s8* i *s9*. Skończyło się problemami z Vaultem opisanymi w dalszych rozdziałach.

7.4 Kolejne próby uruchamiania klastra z maszyny *s2*

Dalsze testy przeprowadzałem na maszynach: *s4*, *s5*, *s6*, *s8* i *s9*.

Najwięcej czasu spędziłem na rozwiązaniu problemu z DNSami opisanym wyżej.

Generowanie inventory z HashiCorp Vault'em

Skrypt *inventory_builder.py* z *Kubespray* generuje wpisy oznaczające węzły jako posiadające HashiCorp Vaulta.

Uruchomienie z Vault'em zakończyło się błędem, więc wyłączyłem Vault'a rozbijając skrypt *bin/setup-cluster-full* na krok konfiguracji i krok uruchomienia, pomiędzy którymi mogłem wyedytować *inventory/inventory.cfg*:

```
1 bin/setup-cluster-configure 10.146.255.{4,5,6,8,9}
2 bin/setup-cluster
```

Próbowałem dostosować parametr *cert_management*, żeby działał zarówno z *Vaultem* jak i bez, ale nie dało to żadnego skutku. Objawem było nie uruchamianie się *etcd*.

Uznałem, że taka konfiguracja jeszcze nie działa i zarzuciłem dalsze próby. Aby rozwiązać problem trzeba usunąć wpisy pod kategorią *[vault]* z pliku *inventory.cfg*.

Niepoprawne znajdowanie adresów IP w ansible

Z jakiegoś powodu konfiguracje *s6* (node3) i *s8* (node4) kończyły się błędem:

```
1 TASK [kubernetes/preinstall : Stop if ip var does not match
   ↪ local ips] *****
2 Friday 26 January 2018 16:37:48 +0100 (0:00:01.297)
   ↪ 0:00:48.587 *****
3 fatal: [node4]: FAILED! => {
```

```

4     "assertion": "ip in ansible_all_ipv4_addresses",
5     "changed": false,
6     "evaluated_to": false
7 }
8 fatal: [node3]: FAILED! => {
9     "assertion": "ip in ansible_all_ipv4_addresses",
10    "changed": false,
11    "evaluated_to": false
12 }

```

Trzy dni później nie wprowadzając po drodze żadnych zmian uruchomiłem klaster bez problemu.

Przyczyną błędu okazały się pozostałości konfiguracji maszyn niezależne ode mnie.

Dostęp do *Kubernetes Dashboard*

Kubernetes Dashboard jest dostępny pod poniższą ścieżką HTTP:

```

1 /api/v1/namespaces/kube-system/services/https:kubernetes-
  ↪ dashboard:/proxy/#!/service/default/kubernetes

```

Można się do niego dostać na dwa sposoby:

1. `kubectl proxy`, które wystawia dashboard na adresie `http://127.0.0.1:8001`
2. Pod adresem `https://10.146.225.4:6443`, gdzie `10.146.225.4` to adres IP dowolnego mastera, w tym przypadku maszyny `s4`

Kompletne adresy to:

```

1 http://127.0.0.1:8001/api/v1/namespaces/kube-system/
  ↪ services/https:kubernetes-dashboard:/proxy/#!/service
  ↪ /default/kubernetes
2 https://10.146.225.4:6443/api/v1/namespaces/kube-system/
  ↪ services/https:kubernetes-dashboard:/proxy/#!/service
  ↪ /default/kubernetes

```

Przekierowanie portów Jeżeli nie pracujemy z maszyny uczelnianej porty możemy przekierować przez SSH na następujące sposoby (jeżeli skrypty uruchamialiśmy z maszyny `s2` i łączymy się do mastera na maszynie `s4`):

1. Plik `~/.ssh/config`:

```

1 Host s2
2   LocalForward 127.0.0.1:8001 localhost:8001
3   LocalForward 127.0.0.1:6443 10.146.225.4:6443

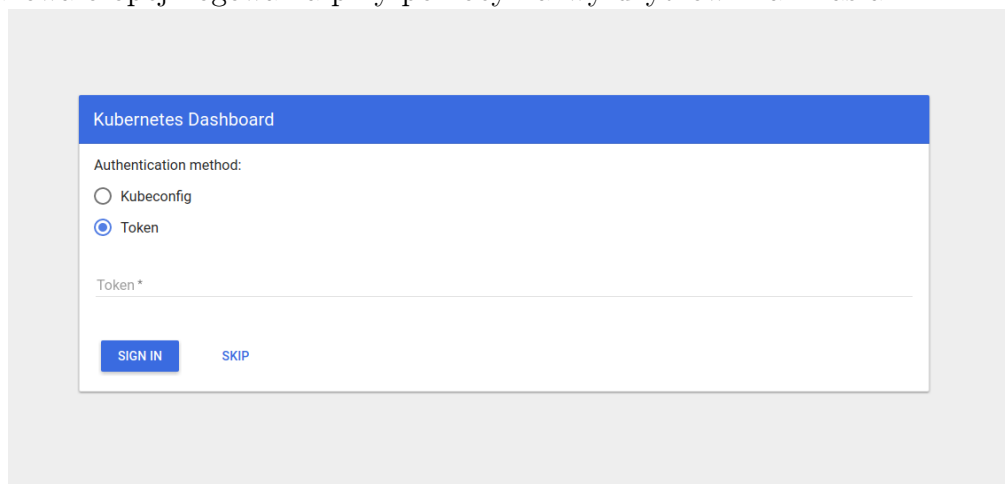
```


2. Argumenty ssh, np.:

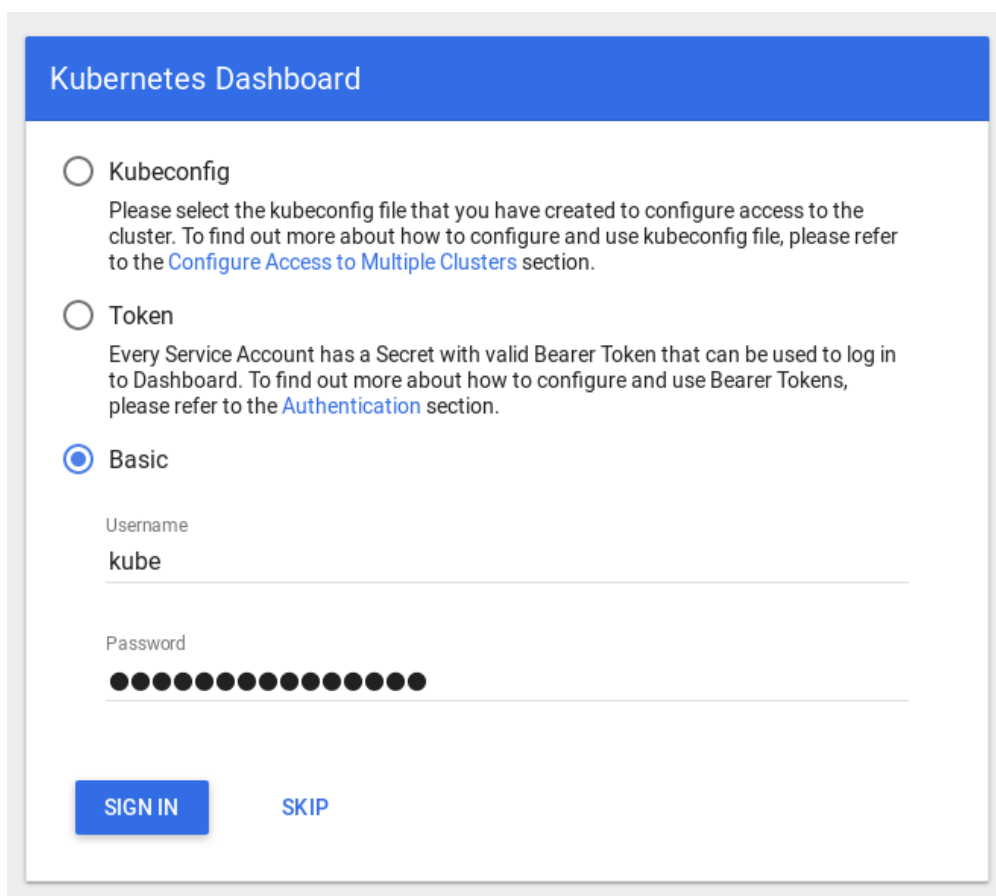
```
1 ssh -L 8001:localhost:8001 -L 6443:10.146.225.4:6443  
    ↪ nazarewk@s2
```

Użytkownik i hasło Domyślna nazwa użytkownika Dashboardu to *kube*, a hasło znajduje się w pliku *credentials/kube_user*.

W starszej wersji (uruchamianej wcześniej) *Kubernetes* i/lub *Kubespary* brakowało opcji logowania przy pomocy nazwy użytkownika i hasła:



Od 29 stycznia 2018 roku widzę poprawny ekran logowania (opcja Basic):

The image shows the Kubernetes Dashboard login interface. At the top is a blue header with the text "Kubernetes Dashboard". Below the header, there are three radio button options for authentication: "Kubeconfig", "Token", and "Basic". The "Basic" option is selected. Under "Kubeconfig", there is a paragraph explaining that the user should select a kubeconfig file and refer to the "Configure Access to Multiple Clusters" section. Under "Token", there is a paragraph explaining that every Service Account has a Secret with a valid Bearer Token and refers to the "Authentication" section. Under "Basic", there are two input fields: "Username" with the value "kube" and "Password" which is masked with dots. At the bottom, there are two buttons: "SIGN IN" in a blue box and "SKIP" in a blue text link.

Instalacja dodatkowych aplikacji z użyciem Kubespray

Kubespray ma wbudowaną instalację kilku dodatkowych aplikacji playbookiem *upgrade-cluster.yml* z tagiem *apps* (skrypt *bin/setup-cluster-upgrade*).

Zmieniłem *kube_script_dir* na lokalizację z poza */usr/local/bin*, bo w systemie *CoreOS* jest read-only squashfs, wybrałem */opt/bin* ponieważ znajdował się już w *PATH*ie na *CoreOS*. Później dowiedziałem się, że domyślnie zmiany *CoreOS* powinny być umieszczane w folderze */opt*

W końcu ze względu na liczne błędy zarzuciłem temat.

Instalacja *Helm*

Helm jest menadżerem pakietów dla *Kubernetes*. Jego głównym zadaniem jest standaryzacja, automatyzacja i ułatwienie instalacji aplikacji w *Kubernetes*.

Helm składa się z:

- programu *helm* uruchamianego lokalnie i korzystającego z danych dostępowych *kubectla*,
- aplikacji serwerowej *Tiller*, z którą *helm* prowadzi interakcje,
- pakietów *Charts* i ich repozytoriów, domyślnie jest to *kubernetes/charts*,

Jego instalacja sprowadza się do:

1. ściągnięcia pliku wykonywalnego dla obecnej architektury,
2. dodania roli RBAC dla *Tillera*,
3. Wywołanie komendy `helm init --service-account tiller`

Wszystkie kroki zawierają się w skrypcie *bin/install-helm*. Ze względu na brak dystrybucji *Helm* na *FreeBSD* całość uruchamiam przez SSH na węźle-zarządcy (domyślnie *s4*).

Szybko okazało się, że większość pakietów wymaga trwałych zasobów dyskowych i nie uda się ich uruchomić bez ich konfiguracji w sieci uczelnianej.

Rozdział 8

Docelowa konfiguracja w sieci uczelnianej

Pełną konfiguracją *Kubernetesa* można uruchomić z maszyny *ldap*; znajduje się ona w folderze */pub/Linux/CoreOS/zetis/kubernetes* maszyny *ldap*, który zawiera podane foldery:

- *kubernetes-cluster* - moje repozytorium zawierające konfigurację i skrypty pozwalające uruchomić klastę,
- *boot* - skrót do folderu *kubernetes-cluster/zetis/WWW/boot* zawierającego konfigurację iPXE oraz Ignition:
 - *coreos.ign* - plik konfigurujący CoreOS, wygenerowany z pliku *coreos.yml* narzędziem do transpilacji konfiguracji *ct*, narzędzie domyślnie nie jest skompilowane na FreeBSD i musimy uruchomić je z Linuxa,
- *log* - standardowe wyjście uruchamianych komend,

8.1 Procedura uruchomienia klastra

1. Wchodzę na maszynie *ldap* do folderu */pub/Linux/CoreOS/zetis/kubernetes/kubernetes-cluster*
2. Upewniam się, że mój klucz SSH znajduje się w *boot/coreos.ign*,
3. Włączam maszyny-węzły wybierając z menu *iPXE CoreOS -> Kubernetes* lub wybierając w narzędziu *boot* bezpośrednio *coreos kub*,
4. Upewniam się, że mam bezhasłowy dostęp do tych maszyn, minimalna konfiguracja *~/.ssh/config* to:

```

1 Host s?
2   User admin
3
4 Host *
5   User nazarewk
6   IdentityFile ~/.ssh/id_rsa
7   IdentitiesOnly yes
8
9 Host s? 10.146.225.*
10  StrictHostKeyChecking no
11  UserKnownHostsFile /dev/null

```

5. Upewniam się, że istnieje folder *kubespray/my_inventory*, jeżeli nie, to go tworzę kopiując domyślną konfigurację:

```

1 cp -rav kubespray/inventory kubespray/my_inventory

```

6. Otwieram plik *inventory/inventory.cfg* i upewniam się, że uruchomione maszyny są obecne w sekcji *[all]* oraz przypisane do odpowiednich ról: *[kube-master]* i *[etcd]* lub *[kube-node]*. Identyfikatorem maszyny jest pierwsze słowo w grupie *[all]*, przykładowa konfiguracja dla maszyn *s4*, *s5* i *s6* z jednym zarządcą to:

```

1 [all]
2 ;s3 ip=10.146.225.3
3 s4 ip=10.146.225.4
4 s5 ip=10.146.225.5
5 s6 ip=10.146.225.6
6 ;s7 ip=10.146.225.7
7 ;s8 ip=10.146.225.8
8 ;s9 ip=10.146.225.9
9 ;sa ip=10.146.225.10
10 ;sb ip=10.146.225.11
11 ;sc ip=10.146.225.12
12
13 [kube-master]
14 s4
15
16 [kube-node]
17 s5
18 s6
19
20 [etcd]
21 s4
22
23 [k8s-cluster:children]
24 kube-node
25 kube-master

```

Opcjonalnie można do każdego węzła:

- dopisać `ansible_python_interpreter=/opt/bin/python`, żeby ułatwić uruchamianie ansibla partiami,
- dopisać `ansible_host=`, jeżeli chce się korzystać z pierwszego wyrazu opisu węzła jako aliasu, a nie faktycznej jego nazwy w sieci uczelnianej,

7. Upewniam się, że plik `inventory/group_vars/all.yml` zawiera naszą konfigurację; minimalny przykład:

```
1 cluster_name: zetis-kubernetes
2 bootstrap_os: coreos
3 kube_basic_auth: true
4 kubeconfig_localhost: true
5 kubectl_localhost: true
6 download_run_once: true
7 cert_management: "{ 'vault' if groups.get('vault', None)
  ↳ else 'script' } }"
8 helm_enabled: true
9 helm_deployment_type: docker
10 kube_script_dir: /opt/bin/kubernetes-scripts
```

8. Uruchamiam konfigurowanie maszyn `bin/setup-cluster` lub bez skryptu:

```
1 ldap% cd kubespray
2 ldap% ansible-playbook -i my_inventory/inventory.cfg
  ↳ cluster.yml -b -v
```

Po około 10-20 minutach skrypt powinien zakończyć się wpisami pokroju:

```
1 ...
2 PLAY RECAP *****
3 localhost                : ok=2      changed=0
  ↳ unreachable=0    failed=0
4 s4                       : ok=281   changed=94
  ↳ unreachable=0    failed=0
5 s5                       : ok=346   changed=80
  ↳ unreachable=0    failed=0
6 s6                       : ok=186   changed=54
  ↳ unreachable=0    failed=0
7 ...
```

9. Weryfikuję instalację:

```
1 ldap% bin/kubectl get nodes
2 NAME      STATUS    ROLES    AGE      VERSION
3 s4        Ready    master   2m       v1.9.1+coreos.0
4 s5        Ready    node     2m       v1.9.1+coreos.0
5 s6        Ready    node     2m       v1.9.1+coreos.0
```

8.2 Sprawdzanie, czy klaster działa

Wywołanie skryptu `bin/students nazarewk create` jest równoważne uruchomieniu komendy `kubectl create -f nazarewk.yml`, gdzie plik `nazarewk.yml` to:

```
1 apiVersion: v1
2 kind: Namespace
3 metadata:
4   name: nazarewk
5   labels:
6     name: nazarewk
7 ---
8 apiVersion: v1
9 kind: ServiceAccount
10 metadata:
11   name: nazarewk
12   namespace: nazarewk
13 ---
14 kind: RoleBinding
15 apiVersion: rbac.authorization.k8s.io/v1
16 metadata:
17   name: nazarewk-admin-binding
18   namespace: nazarewk
19 roleRef:
20   kind: ClusterRole
21   name: admin
22   apiGroup: rbac.authorization.k8s.io
23 subjects:
24 - kind: ServiceAccount
25   name: nazarewk
```

W skrócie:

- tworzę *Namespace*
- tworzę *ServiceAccount*
- przypisuję wbudowaną *Role* o nazwie *admin* do *ServiceAccount* o nazwie *nazarewk* za pomocą *RoleBinding*,

Korzystanie z klastra jako student

- tworzę użytkownika z jego własnym *Namespace*

```
1 ldap% bin/students nazarewk create
2 namespace "nazarewk" created
3 serviceaccount "nazarewk" created
4 rolebinding "nazarewk-admin-binding" created
5 Tokens:
```

```

6 eyJhbGw<<<SKROCONY TOKEN>>>ahHfxU-TRw
7 ldap% bin/students
8 NAME          STATUS    AGE
9 default       Active    3m
10 kube-public   Active    3m
11 kube-system   Active    3m
12 nazarewk      Active    16s

```

- kopiuję token na *s2* z uruchomionym ubuntu:

```

1 ldap% bin/student-tokens nazarewk | ssh nazarewk@s2 "cat
  ↪ > /tmp/token"

```

- pobieram *kubectl*

```

1 s2% cd /tmp
2 s2% curl -LO https://storage.googleapis.com/kubernetes-
  ↪ release/release/$(curl -s https://storage.googleapis.
  ↪ com/kubernetes-release/release/stable.txt)/bin/linux/
  ↪ amd64/kubectl
3 sw% chmod +x kubectl
4 s2% sudo mv kubectl /usr/local/bin
5 s2% source <(kubectl completion zsh)

```

- sprawdzam, czy mam dostęp do klastra

```

1 s2% kubectl get nodes
2 The connection to the server localhost:8080 was refused -
  ↪ did you specify the right host or port?

```

- konfiguruje *kubectl* (najprościej aliasem)

```

1 s2% alias kubectl='command kubectl -s "https://s4:6443" --
  ↪ insecure-skip-tls-verify=true --token="$(cat /tmp/
  ↪ token)" -n nazarewk'

```

- weryfikuję brak dostępu do zasobów globalnych

```

1 s2% kubectl get nodes
2 Error from server (Forbidden): nodes is forbidden: User "
  ↪ system:serviceaccount:nazarewk:nazarewk" cannot list
  ↪ nodes at the cluster scope

```


- tworzę deployment z przykładową aplikacją

```

1 s2% kubectl run echoserver \
2 --image=gcr.io/google_containers/echoserver:1.4 \
3 --port=8080 \
4 --replicas=2
5 deployment "echoserver" created
6 s2% kubectl get deployments
7 NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
8 echoserver     2         2         2            2
9 s2% kubectl get pods
10 NAME          READY   STATUS    RESTARTS
11 echoserver-7b9bbf6ff-22df4  1/1     Running   0
12 echoserver-7b9bbf6ff-c6kbv  1/1     Running   0

```

- wystawiam port, żeby dostać się do aplikacji spoza klastra

```

1 s2% kubectl expose deployment echoserver --type=NodePort
2 service "echoserver" exposed
3 s2% kubectl describe services/echoserver | grep -e NodePort
4 NodePort:          <unset> 30100/TCP
5 s2% curl s4:30100
6 CLIENT VALUES:
7 client_address=10.233.107.64
8 command=GET
9 real path=/
10 query=nil
11 request_version=1.1
12 request_uri=http://s4:8080/
13
14 SERVER VALUES:
15 server_version=nginx: 1.10.0 - lua: 10001
16
17 HEADERS RECEIVED:
18 accept=/*/*
19 host=s4:30100
20 user-agent=curl/7.47.0
21 BODY:
22 -no body in request-

```

- sprawdzam, czy z *ldapa* też mam dostęp do aplikacji:

```

1 ldap% curl s4:30100
2 CLIENT VALUES:
3 client_address=10.233.107.64
4 command=GET
5 real path=/
6 query=nil
7 request_version=1.1
8 request_uri=http://s4:8080/
9
10 SERVER VALUES:
11 server_version=nginx: 1.10.0 - lua: 10001
12
13 HEADERS RECEIVED:
14 accept=/*/*
15 host=s4:30100
16 user-agent=curl/7.58.0
17 BODY:
18 -no body in request-

```

- usuwam użytkownika

```

1 ldap% bin/students nazarewk delete
2 namespace "nazarewk" deleted
3 serviceaccount "nazarewk" deleted
4 rolebinding "nazarewk-admin-binding" deleted
5 Tokens:
6 Error from server (NotFound): serviceaccounts "nazarewk"
  ↪ not found

```

- sprawdzam, czy coś zostało po koncie użytkownika

```

1 ldap% curl s4:30100
2 curl: (7) Failed to connect to s4 port 30100: Connection
  ↪ refused
3 ldap% bin/kubectl get namespace
4 NAME          STATUS    AGE
5 default       Active   46m
6 kube-public   Active   46m
7 kube-system   Active   46m

```

Rozdział 9

Rezultaty i wnioski

Główne założenia pracy inżynierskiej zostały spełnione. Wyjaśniłem dużą ilość zagadnień związanych z *Kubernetes* oraz oddałem do użytku skrypty konfigurujące klaster *Kubernetes* wraz z prostym w obsłudze dodawaniem i usuwaniem jego użytkowników.

W trakcie pisania pracy temat okazał się zbyt obszerny, żeby go kompletnie i wyczerpująco opisać w pracy inżynierskiej. W związku z tym musiałem wybrać tylko najważniejsze informacje i przekazać je w możliwie najkrótszej formie.

Projekt jest bardzo aktywnie rozwijany, więc wiele informacji wyszło na jaw w końcowych etapach pisania pracy. W samej pracy pojawiły się jedynie wzmianki o nich bez dogłębnej analizy.

Nie udało mi się przeprowadzić testów wydajnościowych klastra ze względu na brak czasu.

Dodatek A

Wykaz skryptów

A.1 ipxe-boot

ipxe-boot jest stworzonym przeze mnie na potrzeby pracy repozytorium git skryptów pozwalających na bezdyskowe uruchamianie maszyn poza siecią uczelnianą.

Nie zostały one wykorzystane w końcowej formie pracy inżynierskiej, więc nie będę ich bezpośrednio wymieniał.

A.2 kubernetes-cluster

kubernetes-cluster jest repozytorium Git, które zawiera kompletny kod źródłowy wykorzystywany w tej pracy inżynierskiej i umieszczony w katalogu `/pub/Linux/CoreOS/zetis/kubernetes/kubernetes-cluster/` sieci uczelnianej.

bin/pull

Pobiera aktualną wersję repozytorium wraz z najnowszą wersją zależności:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: git pull wrapper including submodule updates
4
5 git pull || (git fetch --all && git reset --hard origin)
6 git submodule update --init --remote --recursive
7 chmod +x ${0%/*}/*
```

bin/vars

Zawiera wspólne zmienne środowiskowe wykorzystywane w reszcie skryptów oraz linii komend:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: Holds common variables used in all other scripts
4
5 if [ -z "$KC_CONFIGURED" ]; then
6     KC_CONFIGURED=1
7
8     BIN="$(realpath ${0%/*})"
9     PATH="$BIN:$PATH"
10    PDIR="${BIN%/*}"
11    VENV="$PDIR/.env"
12    KUBESPRAY="$PDIR/kubespray"
13    INVENTORY="$KUBESPRAY/my_inventory"
14    CONFIG_FILE="$INVENTORY/inventory.cfg"
15
16    export KUBECONFIG="$KUBESPRAY/artifacts/admin.conf"
17    export ANSIBLE_CONFIG="$PDIR/ansible.cfg"
18 fi
```

bin/ensure-virtualenv

Konfiguruje środowisko Pythona, włącznie z próbą instalacji brakującego *virtualenv* przez *apt*:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: installs python virtualenv
4
5 . ${0%/*}/vars
6 python="$(which python)"
7 activate="$VENV/bin/activate"
8
9 find_virtualenv() {
10     which virtualenv 2> /dev/null || which virtualenv
11 }
12
13 install_virtualenv () {
14     # Installs pip and/or virtualenv
15
16     local pip=$(which pip)
17     local apt=$(which apt)
18     if [ ! -z "$pip" ]; then
19         sudo $pip install virtualenv
20         return 0
21     elif [ ! -z "$apt" ]; then
22         sudo $apt install virtualenv
23         return 0
24     else
25         echo 'Virtualenv, pip and apt-get are both missing,
26             ↪ cannot proceed'
27         exit 1
28     fi
29 }
30
31 create_env () {
32     local cmd=$(find_virtualenv)
33     [ -z "$cmd" ] && install_virtualenv && cmd=$(
34         ↪ find_virtualenv)
35
36     $cmd -p $python $VENV
37     . $activate
38
39     pip install -U pip setuptools
40 }
41
42 [ -z "$python" ] && echo 'python is missing' && exit 1
43 [ -e "$activate" ] && . $activate || create_env
44 pip install -U -r $PDIR/requirements.txt
```

bin/ensure-configuration

Generuje brakujące pliki konfiguracyjne SSH, *inventory.cfg* i *group_vars/all.yml* nie nadpisując istniejących:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: configures cluster (SSH, inventory, group_vars)
4
5 . ${0%/*}/vars
6 SSH_CFG_DST="$HOME/.ssh/config"
7 SSH_CFG="$PDIR/zetis/.ssh/kubernetes.conf"
8
9
10 is_ssh_config_missing () {
11     cat << EOF | python
12 import sys
13 expected = open('$SSH_CFG', 'rb').read()
14 existing = open('$SSH_CFG_DST', 'rb').read()
15 sys.exit(expected in existing)
16 EOF
17     return $?
18 }
19
20 is_ssh_config_missing && cat $SSH_CFG >> $SSH_CFG_DST
21
22 file="$INVENTORY/inventory.cfg"
23 [ -f "$file" ] || cat << EOF > "$file"
24 [all]
25 ;s3 ip=10.146.225.3
26 s4 ip=10.146.225.4
27 s5 ip=10.146.225.5
28 s6 ip=10.146.225.6
29 ;s7 ip=10.146.225.7
30 ;s8 ip=10.146.225.8
31 ;s9 ip=10.146.225.9
32 ;sa ip=10.146.225.10
33 ;sb ip=10.146.225.11
34 ;sc ip=10.146.225.12
35
36 [kube-master]
37 s4
38
39 [kube-node]
40 s5
41 s6
42
43 [etcd]
44 s4
```

```
45
46 [k8s-cluster:children]
47 kube-node
48 kube-master
49 EOF
50
51 file="$INVENTORY/group_vars/all.yml"
52 [ -f "$file" ] || cat << EOF > "$file"
53 cluster_name: zetis-kubernetes
54 bootstrap_os: coreos
55 kube_basic_auth: true
56 kubeconfig_localhost: true
57 kubectl_localhost: true
58 download_run_once: true
59 cert_management: "{ 'vault' if groups.get('vault', None)
    ↪ else 'script' }"
60 helm_enabled: true
61 helm_deployment_type: docker
62 kube_script_dir: /opt/bin/kubernetes-scripts
63 EOF
```


bin/render-coreos

Generuje konfigurację Ignition (JSON) na podstawie Container Linux Config (YAML):

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: renders CoreOS Ignition (incl. retrieving binary
   ↪ )
4
5 . ${0%/*}/vars
6 ct_version=${1:-0.6.1}
7 boot="$PDIR/zetis/WWW/boot"
8 url='https://github.com/coreos/container-linux-config-
   ↪ transpiler'
9 url="$url/releases/download"
10 url="$url/v${ct_version}/ct-v${ct_version}-x86_64-unknown-
   ↪ linux-gnu"
11
12 wget -nc $url -O $BIN/ct
13 chmod +x $BIN/ct
14 $BIN/ct -pretty \
15   -in-file "$boot/coreos.yml" \
16   -out-file "$boot/coreos.ign"
```

bin/setup-cluster

Właściwy skrypt konfiguruje klaster na działających maszynach CoreOS:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: sets up the cluster
4
5 . ${0%/*}/vars
6 . $BIN/ensure-configuration
7 . $BIN/activate
8
9 (
10   cd $KUBESPRAY;
11   ansible-playbook -i $INVENTORY/inventory.cfg cluster.yml
12   ↪ -b -v $@
13 )
14 chmod -R 700 $KUBESPRAY/{artifacts/admin.conf,credentials}
15
16 cat << EOF
17 Login credentials:
18   user: kube
19   password: $(cat $PDIR/credentials/kube_user)
20 EOF
```

bin/setup-cluster-full

Skrót do pobierania najnowszej wersji, a następnie uruchamiania klastra:

```
1 #!/bin/sh
2 . ${0%/*}/vars
3
4 $BIN/pull
5 $BIN/setup-cluster $@
```

bin/setup-cluster-upgrade

Skrypt analogiczny do *setup-cluster*, ale wywołujący *upgrade-cluster.yml* zamiast *cluster.yml*:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: runs upgrade-cluster.yml instead of cluster.yml
4
5 . ${0%/*}/vars
6 . $BIN/ensure-configuration
7 . $BIN/activate
8
9 cd $KUBESPRAY
10 ansible-playbook -i $INVENTORY/inventory.cfg upgrade-
    ↪ cluster.yml -b -v $@
```

bin/kubectl

Skrót *kubectl* z automatycznym dołączaniem konfiguracji *kubespray*:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: kubectl wrapper which uses generated
    ↪ configurations
4
5 # Handle running kubectl from PATH
6 [ "${0%/*}" == "$0" ] \
7 && _bin=$(dirname $(which kubectl)) \
8 || _bin=${0%/*}
9
10 . $_bin/vars
11
12 local_bin="$PDIR/artifacts/kubectl"
13
14 if [ -f $local_bin ]; then
15     chmod +x $local_bin
16     kubectl=$local_bin
17 else
18     kubectl=/usr/bin/kubectl
19 fi
20
21 $kubectl $@
```

bin/students

Skrypt zarządzający obiektami *Kubernetes* użytkowników, czyli studentów:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: executes operations on users (create, get,
4           ↪ describe, delete)
5
6 . ${0%/*}/vars
7
8 kubectl_command () {
9     local users="$1"
10    local cmd="$2"
11    for name in $users; do
12        cat <<EOF | $BIN/kubectl $cmd -f -
13        apiVersion: v1
14        kind: Namespace
15        metadata:
16            name: ${name}
17            labels:
18                name: ${name}
19        ---
20
21        apiVersion: v1
22        kind: ServiceAccount
23        metadata:
24            name: ${name}
25            namespace: ${name}
26        ---
27
28        kind: RoleBinding
29        apiVersion: rbac.authorization.k8s.io/v1
30        metadata:
31            name: ${name}-admin-binding
32            namespace: ${name}
33        roleRef:
34            kind: ClusterRole
35            name: admin
36            apiGroup: rbac.authorization.k8s.io
37        subjects:
38            - kind: ServiceAccount
39              name: ${name}
40        EOF
41    done
42    echo Tokens:
43    $BIN/student-tokens ${name}
```

```

44 done
45 }
46
47 case $1 in
48     ""|list)
49         $BIN/kubectl get namespace
50         ;;
51     -h)
52         cat << EOF
53         Usage: $0 "<usernames>" create|get|describe|delete
54             where <usernames> is a single-argument list of
55                 ↪ usernames
56         You can also call it without arguments to list users (
57                 ↪ namespaces)
58 EOF
59         ;;
60     *)
61         kubectl_command "$@"
62         ;;
63 esac

```

bin/student-tokens

Listuje przepustki konkretnego użytkownika:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: lists kubernetes authentication tokens for
   ↪ specified user
4
5 . ${0%/*}/vars
6 name=$1
7 args="--namespace $name get serviceaccount $name"
8 args="$args -o jsonpath={.secrets[*].name}"
9 for token in `${BIN}/kubectl $args`; do
10     args="get secrets $token --namespace $name"
11     args="$args -o jsonpath={.data.token}"
12     ${BIN}/kubectl $args | base64 --decode
13     echo
14 done
```

bin/install-helm

Instaluje menadżer pakietów *Helm*:

```
1 #!/bin/sh
2 # Author: Krzysztof Nazarewski <nazarewk@gmail.com>
3 # Purpose: installs helm package manager into Kubernetes
4           ↪ cluster
5 . ${0%/*}/vars
6
7 host=${1:-s4}
8
9 ssh () {
10     command ssh $host sudo $@
11 }
12
13 cat <<EOF | ssh bash
14 export HELM_INSTALL_DIR=/opt/bin
15 url='https://raw.githubusercontent.com/kubernetes/helm/
16     ↪ master/scripts/get'
17 [ -f /opt/bin/helm ] || curl \ $url | bash
18 EOF
19
20 cat <<EOF | ssh kubectl create -f -
21 apiVersion: v1
22 kind: ServiceAccount
23 metadata:
24   name: tiller
25   namespace: kube-system
26 ---
27 apiVersion: rbac.authorization.k8s.io/v1beta1
28 kind: ClusterRoleBinding
29 metadata:
30   name: tiller
31 roleRef:
32   apiGroup: rbac.authorization.k8s.io
33   kind: ClusterRole
34   name: cluster-admin
35 subjects:
36   - kind: ServiceAccount
37     name: tiller
38     namespace: kube-system
39 EOF
40 ssh helm version | grep Server: || ssh helm init --service-
41     ↪ account tiller
```

zetis/.ssh/kubernetes.conf

Częściowy plik konfiguracyjny SSH do umieszczenia w *~/.ssh/config*:

```
1 Host s?
2   User admin
3
4 IdentityFile ~/.ssh/id_rsa
5 IdentitiesOnly yes
6
7 Host s?
8   StrictHostKeyChecking no
9   UserKnownHostsFile /dev/null
```


zetis/WWW/boot/coreos.ipxe

Skrypt iPXE uruchamiający maszynę z CoreOS:

```
1 #!/ipxe
2 set ftp http://ftp/pub/
3
4 set base-url ${ftp}/Linux/CoreOS/alpha
5 set ignition ${ftp}/Linux/CoreOS/zetis/kubernetes/boot/
   ↪ coreos.ign
6
7 set opts ${opts} coreos.autologin
8 set opts ${opts} coreos.first_boot=1 coreos.config.url=${
   ↪ ignition}
9 set opts ${opts} systemd.journald.max_level_console=debug
10 kernel ${base-url}/coreos_production_pxe.vmlinuz ${opts}
11 initrd ${base-url}/coreos_production_pxe_image.cpio.gz
12
13 boot
```

zetis/WWW/boot/coreos.yml

Plik konfiguracyjny Container Linux Config w formacie YAML, docelowo do przeprowadzenia przez narzędzie *ct*. Wyjątkowo skróciłem ten skrypt, ze względu na długość kluczy SSH:

```
1 passwd:
2   users:
3     - name: admin
4       groups: [sudo, docker]
5       ssh_authorized_keys:
6         - ssh-rsa AAAAB3N...dAYs7Y6L8= ato@volt.iem.pw.edu.pl
7         - ssh-rsa AAAAB3N...N9aLYp0ct/ nazarewk
8         - ssh-rsa AAAAB3N...XRjw== nazarewk@ldap.iem.pw.edu.pl
```

zetis/WWW/boot/coreos.ign

Z powyższego pliku wygenerowany zostaje (również skrócony) plik JSON:

```
1 {
2   "ignition": {
3     "config": {},
4     "timeouts": {},
5     "version": "2.1.0"
6   },
7   "networkd": {},
8   "passwd": {
9     "users": [
10      {
11        "groups": [
12          "sudo",
13          "docker"
14        ],
15        "name": "admin",
16        "sshAuthorizedKeys": [
17          "ssh-rsa AAAAB3N...N9aLYp0ct/ nazarewk",
18          "ssh-rsa AAAAB3N...XRjw== nazarewk@ldap.iem.pw.
19            ↪ edu.pl",
20          "ssh-rsa AAAAB3N...dAYs7Y6L8= ato@volt.iem.pw.edu
21            ↪ .pl"
22        ]
23      }
24    ],
25    "storage": {},
26    "systemd": {}
27  }
```