

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 pracy jest implementacja klastra Kubernetes w sieci komputerowej bazującej na maszynach bezdyskowych. W pracy przedstawione zostały podstawowe pojęcia związane z kontenerami, systemem Kubernetes i maszynami bezdyskowymi. Część praktyczna zawiera przegląd systemów operacyjnych i dostępnych konfiguratorów klastra Kubernetes, a kończy się prezentacją implementacji klastra Kubernetes w sieci komputerowej Wydziału Elektrycznego Politechniki Warszawskiej.

Słowa kluczowe: Kubernetes, konteneryzacja, zarządzanie kontenerami, systemy bezdyskowe

Implementing cluster Kubernetes based on diskless machines

Abstract

Purpose of the this thesis is implementation of Kubernetes cluster in the network based on diskless systems. The paper presents basic concepts of containers, Kubernetes system and diskless machines. This paper includes practical overview of operating systems and available Kubernetes configurators. Publication ends with a presentation of real world deployment of Kubernetes cluster in the Faculty of Electrical Engineering of Warsaw University of Technology network.

Keywords: Kubernetes, containerization, container 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	Charakterystyka systemów bezdyskowych	2
2	System Kubernetes	4
2.1	Struktura klastra Kubernetes	5
2.2	Obiekty Kubernetes API	9
2.3	Zarządzanie klastrem Kubernetes	13
2.3.1	Wybór narzędzia konfiguracyjnego	14
2.4	Lista alternatyw Kubernetes	15
3	Praktyczne rozeznanie w narzędziach zarządzania klastrem Kubernetes	20
3.1	Rancher 2.0	20
3.2	OpenShift Origin	22
3.3	kubespray	26
4	Wstępna implementacja kubespray w laboratorium 225	27
4.1	Konfiguracja kubespray	28
4.2	Napotkane problemy	30
5	Prezentacja docelowej konfiguracji klastra Kubernetes w sieci uczelnianej	34
5.1	Procedura uruchomienia klastra	34
5.2	Procedury korzystania z klastra Kubernetes	36
6	Podsumowanie	41
A	Wykaz odnośników	42
B	Wykaz kodu źródłowego	44
B.1	repozytorium kubernetes-cluster	44
B.2	repozytorium ipxe-boot	53

Rozdział 1

Wstęp

Historia wirtualizacji i przydzielania zasobów sięga już pierwszych systemów komputerowych.

Podstawową przeszkodą w korzystaniu z komputerów była ich wysoka cena; bogate instytucje odpłatnie wynajmowały swoje systemy innym firmom. Umożliwiając dostęp tylko jednemu użytkownikowi jednocześnie, maszyny przez większość czasu nie pracowały. Wraz ze wzrostem liczby obsługiwanych użytkowników przypadających na jedną maszynę zwiększało się wysycenie jej zasobów, a zatem i zyski.

Następną przeszkodą okazał się brak ograniczeń dostępowych dla poszczególnych użytkowników: każdy świadomie lub nie mógł uszkodzić cudze zasoby. W ten sposób zrodził się pomysł `chroot`, czyli izolacji użytkowników na poziomie systemu plików.

Wraz z rozwojem technologii zwiększała się różnorodność sprzętowa i systemowa. W związku z pierwszą systemy operacyjne oferowały abstrakcje dostępu do sprzętu oraz powstawały narzędzia emulujące inny sprzęt. Różnorodność systemowa była natomiast abstrahowana przez aplikacyjne wirtualne maszyny, które umożliwiały jednnorodny dostęp do zasobów niejednorodnych systemów operacyjnych, ale w żaden sposób go nie ograniczały.

W centrach danych zyskała popularność wirtualizacja sprzętowa. Pozwala ona uruchomić na jednej fizycznej maszynie wiele niezależnych od siebie maszyn wirtualnych, na których działają dowolne systemy operacyjne. W toku rozwoju technologii po roku 2000 pojawił się trend rozpraszania oprogramowania i dążenia do uruchamiania jak najmniejszej jego funkcjonalnej części na pojedynczej maszynie wirtualnej. Główną zaletą takiego podejścia jest bezpieczeństwo: po uzyskaniu dostępu do jednego z serwerów włamywacz nadal nie ma dostępu do reszty infrastruktury. Wadą natomiast jest to, że jedna maszyna fizyczna jest w stanie uruchomić nie więcej niż kilka do kilkunastu maszyn wirtualnych. W związku z tym utrzymywanie całej maszyny

wirtualnej w celu uruchamiania pojedynczego procesu aplikacyjnego często jest nieekonomiczne.

Na przełomie ostatniej dekady zaczęły pojawiać się narzędzia pozwalające uruchamiać wiele w pełni odizolowanych procesów współdzielących jedynie jądro systemu operacyjnego. Znacznie zmniejszyło to narzut zasobów wymaganych do uruchamiania kolejnych aplikacji-procesów i umożliwiło współdzielenie jednej maszyny przez setki, a nawet tysiące programów. Metodologia ta zyskała miano konteneryzacji i jest z powodzeniem wykorzystywana w systemach produkcyjnych na całym świecie.

Technologia konteneryzacji jest młoda, a konkurencja duża. Z jednej strony trwa wyścig o stworzenie najlepszego rozwiązania, a z drugiej aktywnie rozwijane są otwarte standardy konteneryzacji w ramach `Open Container Initiative` \hookrightarrow ¹, ponieważ nikt nie może pozwolić sobie na niekompatybilność swojej implementacji. Obecnie najpopularniejszą implementacją kontenerów jest `Docker` \hookrightarrow .

Aspektem równie ważnym jak sama implementacja kontenerów, jest kwestia zarządzania nimi. O ile kilkoma serwerami z kilkudziesięcioma kontenerami może zarządzać człowiek, o tyle do zarządzania tysiącami kontenerów wymagany jest w dużej mierze zautomatyzowany i wyspecjalizowany system.

Obecnie przodującym rozwiązaniem tego problemu jest `Kubernetes` (zamiennie `k8s`). Projekt powstał niedawno (w połowie 2014 roku) i mimo tego, że coraz bardziej umacnia swoją pozycję na rynku nadal umie go skonfigurować tylko nieliczne grono specjalistów. Celem niniejszej pracy jest przedstawienie sposobu konfiguracji klastra `k8s` na fizycznych maszynach* na przykładzie sieci komputerowej Wydziału Elektrycznego Politechniki Warszawskiej. Laboratoria komputerowe dostępne w tej sieci są wyposażone w maszyny bezdyskowe, a zatem należy pokrótce scharakteryzować tego typu systemy.

1.1 Charakterystyka systemów bezdyskowych

W dzisiejszych czasach rozwój sieci komputerowych i spadek cen pamięci RAM pozwolił na konfigurację maszyn nie posiadających dysku twardego. Bezdyskowe maszyny przy starcie pytają centralny serwer o swoją konfigurację (np. protokołem `PXE`), a następnie umieszczają kompletny system operacyjny w pamięci RAM.

W momencie wyłączenia maszyny cały stan jest tracony, a przy ponownym rozruchu konfigurowana jest od nowa. w celu trwałego przechowywania

*ang. *bare metal*, czyli uruchamianie na prostej sieci komputerowej bez zewnętrznych zależności, w odróżnieniu od kompleksowych usług chmurowych, które w znacznym stopniu automatyzują i ułatwiają konfigurację

danych systemy bezdyskowe korzystają z zewnętrznych serwerów plikowych.

Rozwiązanie pozwala to na jednorazową konfigurację bazowego systemu operacyjnego i uzyskanie konfiguracji na dowolnej ilości maszyn bez ingerencji człowieka. W efekcie koszt utrzymania centrów danych przenosi się z konfiguracji indywidualnych maszyn na konfigurację pojedynczych usług działających na dowolnej liczbie maszyn.

Zarządzanie klastrem `k8s` poza środowiskiem chmurowym znacznie ogranicza wybór narzędzia konfiguracyjnego, a bezdyskowość jego węzłów ma duży wpływ na wybór systemu operacyjnego.

Rozdział 2

System Kubernetes

Kubernetes² został stworzony przez firmę Google na bazie ich wewnętrznego systemu Borg, a następnie udostępniony na zasadach otwartego oprogramowania (ang. *open-source*). Charakteryzuje się korzystnym stosunkiem kosztów zarządzania do oferowanych możliwości.

Pod jest podstawową i niepodzielną jednostką aplikacyjną klastra `k8s` dalej zwaną instancją aplikacji lub po prostu instancją. W rozdziale “Podstawowe rodzaje obiektów aplikacyjnych” znajdują się bardziej szczegółowe informacje dotyczące tego zagadnienia.

Kontroler jest ogólnym określeniem grupy obiektów zarządzających instancjami aplikacji.

Zarządzanie klastrem rozumiem jako zbiór czynności i procesów polegających na przygotowaniu klastra do użytku oraz definiowanie aplikacji realizującymi usługi na poziomie klastra, takimi jak:

- tworzenie klastra,
- dodawanie i usuwanie fizycznych węzłów,
- integracja z sieciowym systemem plikowym,
- integracja z zewnętrznymi systemami zarządzania kontami użytkowników,
- nadawanie uprawnień użytkownikom i procesom,
- centralny system zbierania dzienników aplikacji (ang. *centralized logging system*),
- centralny system monitorujący obciążenie węzłów,

Wykorzystanie klastra rozumiem jako uruchamianie dowolnych aplikacji na wcześniej skonfigurowanym klastrze.

Stan w klastrze Kubernetes zdefiniowany jest jako konfiguracja, do której klaster dąży.

Status w klastrze Kubernetes zdefiniowany jest jako rzeczywista konfiguracja, w której klaster się w danej chwili znajduje.

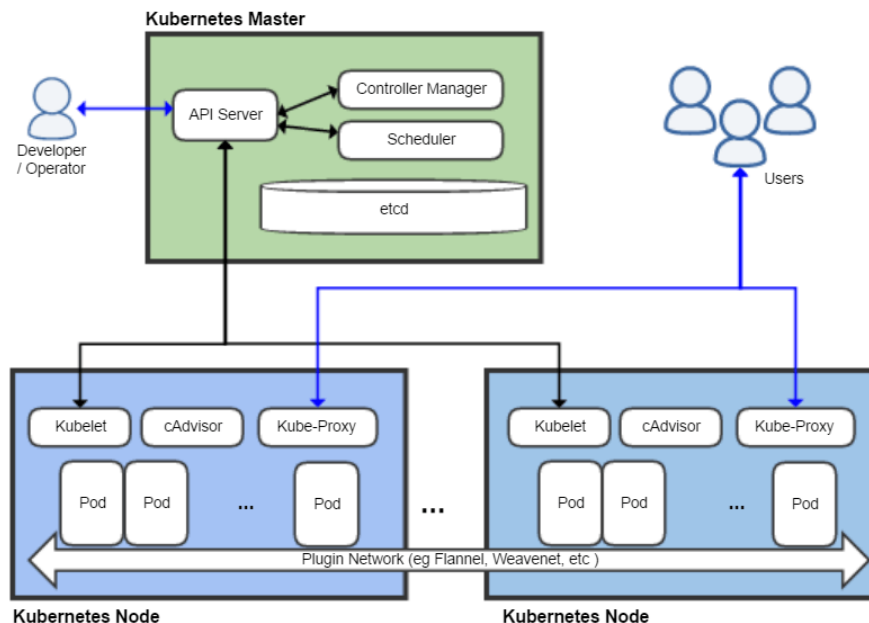
Zwykle stan i status klastra szybko się ze sobą zbiegają, ale nie jest to regułą. Dwoma najczęstszymi powodami braku ich synchronizacji są niewystarczające zasoby lub utrata węzła roboczego.

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 wymagane zasoby i aplikacja zostanie uruchomiona w pełni, czyli stan aplikacji pokryje się z jej statusem.

W drugim przypadku mając działające 4 instancje aplikacji na 2 węzłach, po utraceniu drugiego węzła stan kontrolera będzie nadal wskazywał na 4 instancje, a status na 2. W ciągu kilku do kilkudziesięciu sekund kontroler zauważy braki, uruchomi brakujące 2 instancje na pierwszym węźle i przez to doprowadzi status klastra do stanu docelowego.

2.1 Struktura klastra Kubernetes

Wysokopoziomową strukturę klastra `k8s` przedstawia poniższy schemat:



Na ilustracji wyróżnione zostało 5 grup funkcjonalnych:

1. administrator lub programista korzystający z klastra - Developer / Operator
→ ,
2. użytkowników końcowi aplikacji działających w klastrze - Users ,
3. jeden lub więcej węzeł zarządzający - Kubernetes Master ,
4. jeden lub więcej węzeł roboczy - Kubernetes Node ,
5. wtyczka sieciowa (Plugin Network), czyli oprogramowanie realizujące komunikację siecią w ramach klastra k8s.

Płaszczyzna zarządzająca

Węzeł zarządzający zilustrowany został jako jedna maszyna, ale w trybie wysokiej dostępności składa się z co najmniej trzech kopii. W związku z tym trafniejszą jego nazwą jest “płaszczyzna zarządzająca” (ang. *Control Plane*).

etcd jest bazą danych kluczy i wartości, w której przechowywany jest zarówno stan, jak i status klastra. Jest koncepcyjnie proste, żeby ułatwić skupienie się na jej wydajności, stabilności i skalowaniu. Mimo że najczęściej przedstawia się je jako składową węzła zarządzającego, nie jest to regułą, a tym bardziej wymogiem.

kube-apiserver jest jedynym sposobem na interakcję z etcd w ramach klastra k8s. Zarówno zewnętrzni użytkownicy, jak i wewnętrzne procesy klastra korzystają z interfejsu aplikacyjnego REST (ang. REST API) oferowanego przez kube-apiserver w celu uzyskania informacji i zmiany jego stanu.

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

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

Węzeł roboczy

W klastrze k8s istnieją tylko dwa niezbędne elementy węzła roboczego, ale w praktyce każdy z węzłów ma uruchomiony cały wachlarz aplikacji ułatwiających zarządzanie nimi. Zilustrowanym przykładem jest cAdvisor.

kubelet jest podstawowym procesem działającym na węzłach roboczych. 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, dodawać nowe lub wyłączać nadwyżkowe.

kube-proxy jest drugim najważniejszym procesem węzła roboczego odpowiadającym za przekierowywanie ruchu sieciowego do odpowiednich kontenerów w ramach klastra.


cAdvisor jest opcjonalnym elementem węzła roboczego, który monitoruje zużycie zasobów i wydajność kontenerów w ramach jednego klastra. Umożliwia automatyczne skalowanie aplikacji, na przykład na podstawie dużego obciążenia pamięci RAM lub procesora.

Komunikacja sieciowa

Podstawowym założeniem k8s jest posiadanie własnego adresu IP przez każdą aplikację działającą w klastrze, ale nienarzucanie rozwiązania je realizującego to założenie. Administrator klastra musi zadbać o konfigurację

odpowiednich narzędzi, zwanych w nomenklaturze `k8s` *overlay network*. Najprostsze koncepcyjnie jest stworzenie na każdym węźle wpisów `iptables` przekierowujących adresy IP na wszystkie inne węzły, a najpopularniejszymi rozwiązaniami są: `Flannel`³ i `Project Calico`⁴.

W klastrze `k8s` istnieją 4 podstawowe rodzaje komunikacji sieciowej:

1. wewnątrz kontenerów należących do jednej instancji na adresie `127.0.0.1`  ,
2. pomiędzy instancjami w klastrze realizowana przez `overlay network` ,
3. między instancjami i serwisami realizowana przez `kube-proxy` ,
4. między serwisami, a światem realizowana serwisami typu `LoadBalancer` lub mechanizmem `Ingress` ,

Poza nakładką sieciową realizującą połączenie między kontenerami (w ramach adresu `127.0.0.1`) składającymi się na aplikację istnieją jeszcze komunikacja aplikacji z serwisami i serwisami z siecią zewnętrzną.

Serwis (ang. *Service*) jest podstawowym sposobem na zautomatyzowanie komunikacji z instancjami aplikacji zarówno w obrębie klastra jak i z poza niego. Podstawowymi rodzajami serwisów są:

- `ClusterIP` udostępniający instancję na wewnętrznym adresie IP,
- `NodePort` udostępniający instancję na zewnętrznym adresie IP i porcie węzła na którym został uruchomiony,
- `LoadBalancer` konfiguruje zewnętrzną usługę przekierowującą zapytania do automatycznie tworzonego serwisu typu `ClusterIP` lub `NodePort` ,

Ingress jest mechanizmem umożliwiającym eksponowanie instancji poza klastery, tak jak ma to miejsce w serwisie typu `LoadBalancer` , ale dającym większe możliwości konfiguracji, na przykład eliminując zależność od usług chmurowych. Obecnie istnieją kontrolery `Ingress` bazujące na serwerze `nginx`⁵ i `GCE`⁶.

Zarządzanie dostępami

Podstawowymi pojęciami związanymi z zarządzaniem dostępami w `k8s` są uwierzytelnianie, autoryzacja i przestrzenie nazw (obiekt `Namespace`).

Uwierzytelnianie jest pierwszym krokiem w każdym zapytaniu do API. Polega na weryfikacji, że użytkownik (lub wewnętrzna 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,
- zapytanie uwierzytelniające do zewnętrznego serwisu (ang. *webhook*),
- proxy uwierzytelniające.

Ze względu na prostotę i uniwersalność rozwiązania w pracy zostaną wykorzystane tylko konta serwisowe.

Autoryzacja jest drugim etapem zapytania do API. Polega na weryfikacji, że użytkownik jest uprawniony do korzystania z danego zasobu.

Najpopularniejszym sposobem autoryzacji jest RBAC⁷ (ang. *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`.

Przestrzeń nazw (obiekt `k8s Namespace`) jest logicznie odseparowaną częścią klastra. Pozwala na współdzielenie jednego klastra przez wielu niezauważanych użytkowników. Standardowym zastosowaniem jest wydzielanie środowisk produkcyjnych, QA i deweloperskich.

Jak sama nazwa wskazuje role z dopiskiem `cluster` mogą dać dostęp do wszystkich przestrzeni nazw jednocześnie oraz zasobów, które ich nie mają. Przykładem zasobu nieposiadającego swojej przestrzeni nazw jest węzeł (`Node` →) lub zakończenie API `/healthz`.

Role bez dopisku `cluster` operują w ramach jednej przestrzeni nazw.

2.2 Obiekty Kubernetes API

Architektura klastra `k8s` bazuje na interfejsie REST (ang. *Representational State Transfer*), czyli zmianie stanu przez reprezentację (obiekty). Wszystkie mechanizmy (obiekty deklaratywne) i akcje (obiekty imperatywne) dostępne w klastrze mają swoje odzwierciedlenie w obiektach `Kubernetes`. Szczegółowy opis konwencji API obiektów możemy znaleźć w dokumencie z repozytorium `k8s`⁸.

Użytkownicy klastra operują na reprezentacji obiektów w formacie `YAML` lub `JSON`, przykładowy obiekt wygląda następująco:

```

apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: my-namespace
  uid: 343fc305-c854-44d0-9085-baed8965e0a9
  labels:
    resources: high
  annotations:
    app-type: qwe
spec:
  containers:
  - image: alpine
    command: ["echo"]
    args: ["Hello World"]
  ...
status:
  podIP: 127.12.13.14
  ...

```

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

- `apiVersion`: obowiązkowa wersja API k8s, np. `v1` lub `v1beta1`,
- `kind`: obowiązkowy typ obiektu zdefiniowanego w specyfikacji wersji API,
- `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 status obiektu,

Podstawowe rodzaje obiektów aplikacyjnych

Pod reprezentuje nierozłącznie powiązaną (np. współdzielonymi zasobami) grupę jednego lub więcej kontenerów. W odróżnieniu od innych obiektów jest

aktualnie działającą aplikacją, czyli reprezentuje wyżej zdefiniowany status klastra.

Instancje są bezustannie uruchamiane i wyłączane przez kontrolery. Trwałość danych można uzyskać przydzielając im sieciowe zasoby dyskowe, podobnie jak ma to miejsce w przypadku maszyn bezdyskowych. Instancje aplikacji nie powinny być zarządzane bezpośrednio, jedynie przez kontrolery. Najczęściej konfigurowane są przez `PodTemplateSpec`, czyli szablony ich specyfikacji.

ReplicaSet jest następcą `ReplicaController`, czyli imperatywnym kontrolerem dbającym o działanie określonej liczby `Pod`ów w klastrze.

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

Deployment pozwala na deklaratywne aktualizacje `Pod` i `ReplicaSet`. Korzystanie z nich bezpośrednio nie jest zalecane.

Zmiany `Deployment` wprowadzane są przez tak zwane `rollout`. Każdy ma swój status i może zostać wstrzymany lub przerwany. `Rollout` może zostać aktywowany automatycznie przez zmianę `.spec.template`. `Deployment` posiada swoją historię, w formie rewizji, która są zmieniane w trakcie `rollout`. Operacja skalowania nie uruchamia `rollout`, a więc nie zmienia rewizji.

Podstawowe przypadki użycia `Deployment` to:

- uruchamianie `ReplicaSet` w tle przez `.spec.replicas`,
- deklarowanie nowego stanu instancji poprzez zmianę `.spec.template`,
- cofanie zmian do poprzednich rewizji `Deployment` komendą `kubectl rollout undo`,
- automatyczne skalowanie `Deployment` w celu obsługi większego obciążenia, np. `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` przez ograniczanie liczby wpisów w `.spec.revisionHistoryLimit`.


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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
```

```

selector:
  matchLabels:
    app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80

```

Pole `.spec.selector` definiuje w jaki sposób Deployment ma znaleźć Pod , którym ma zarządzać. Selektor powinien zgadzać się ze zdefiniowanym szablonem.

StatefulSet jest kontrolerem podobnym do Deployment umożliwiającym zachowanie stanu aplikacji między uruchomieniami.

W przeciwieństwie do Deployment StatefulSet nadaje każdej uruchomionej aplikacji stały unikalny identyfikator, który zostaje zachowany mimo wyłączenia i przenoszenia instancji. Identyfikatory można zastosować między innymi do:

- trwałych i unikalnych identyfikatorów wewnątrz sieci,
- trwałych zasobów dyskowych,
- sekwencyjnego uruchamiania i skalowania aplikacji,
- sekwencyjnego wygaszania i usuwania aplikacji,
- sekwencyjnych, zautomatyzowanych aktualizacji aplikacji.

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 pozwala na jednorazowe uruchomienie instancji aplikacji, które wykonują akcję i się kończą. Domyślnie przy niepowodzeniu uruchamiane są kolejne Pody aż zostanie uzyskana odpowiednia liczba sukcesów.


CronJob pozwala na generowanie Job jednorazowo o określonym czasie lub cyklicznie zgodnie ze specyfikacją `cron`⁹.

2.3 Zarządzanie klastrem Kubernetes

Ze względu na dużą liczbę dostępnych narzędzi konfiguracyjnych wymienione zostały tylko wybrane.

kubectl jest podstawowym narzędziem prowadzącym interakcję z klastrem, łączy się do serwera API i modyfikuje stan klastra.

Helm jest menadżerem pakietów dla `k8s`. Jego głównymi zadaniami są: standaryzacja, automatyzacja i ułatwienie instalacji aplikacji w `k8s`. Składa się z trzech części:

- programu `helm` uruchamianego lokalnie i korzystającego z danych dostępowych `kubectl`,
- aplikacji serwerowej `Tiller`, z którą `helm` prowadzi interakcje,
- pakietów `Charts` i ich repozytoriów, domyślnie jest to `kubernetes / charts` ¹⁰,

Narzędzia ograniczone do linii komend

kops jest oficjalnie wspieranym, a przez to najpopularniejszym konfiguratorem klastra `k8s`, ale jak konfiguratorów zakłada uruchomienie w środowiskach chmurowych, `PaaS` lub `IaaS`. W związku z tym nie ma żadnego zastosowania w tej pracy inżynierskiej.

kubeadm jest oficjalnie wspieranym, narzędziem pozwalającym na niskopoziomowe zarządzanie klastrem `k8s`. Wiele innych konfiguratorów na nim bazuje.

kubespray jest konfiguratorem bazującym na `Ansible`, stanowi kontynuację projektu `kargo`. Charakteryzuje się wspieraniem szerokiej gamy konfiguracji, w tym sprzętowego klastra `Kubernetes`. Projekt obecnie znajduje się w inkubatorze `k8s` i jest aktywnie rozwijany. Istnieje narzędzie o nazwie `kubespray -cli` ułatwiające korzystanie z właściwego `kubespray`, ale nie nadało za głównym projektem i zostało porzucone.

Canonical distribution of Kubernetes jest prostą w instalacji dystrybucją `k8s` od firmy Canonical. Wymaga przynajmniej podstawowej infrastruktury chmurowej, w celu uruchomienia klastra składającego się z więcej niż jednego węzła. Klaster sprzętowy bazuje na systemie `Metal as a Service`¹¹, w związku z tym nie będzie dalej zajmował się tym narzędziem.

Typhoon Projekt `Typhoon` bazując na narzędziu `bootkube` realizuje konfigurację sprzętowego klastra `k8s`¹² w trakcie rozruchu systemu operacyjnego `CoreOS`. Domyślnie `Typhoon` instaluje `k8s` na dysku, a następnie restartuje maszynę. W dokumentacji pojawiła się krótka wzmianka o możliwości instalacji bezdyskowej, ale nie zostały podane żadne wskazówki, w związku z tym narzędzie zostało odrzucone.

Narzędzia graficzne

Kubernetes Dashboard jest podstawowym interfejsem graficznym klastra `k8s` dostępnym w każdej jego dystrybucji. Umożliwia monitorowanie, analizę błędów i podstawową zmianę stanu klastra.

Rancher jest platformą zarządzania kontenerami umożliwiającą między innymi zarządzanie klastrem `k8s`. Od wersji 2.0 twórcy skupiają się wyłącznie na zarządzaniu `k8s` porzucając wsparcie innych rozwiązań.

OpenShift jest komercyjną usługą typu `PaaS` udostępnianą przez firmę Red Hat. Od wersji 3 skupia się na zarządzaniu klastrem `k8s`. Rdzeniem projektu jest konfigurator `OpenShift Origin`¹³ bazujący na `OpenShift Ansible`.

DC/OS jest częścią najbardziej zaawansowanego systemu zarządzania kontenerami `Mesos` utrzymywanym przez fundację Apache. Niedawno został rozszerzony o wykorzystywanie `k8s`¹⁴ jako drugiego systemu zarządzania kontenerami. Do tej pory bazował na `Marathon`, czyli rozwiązaniu konkurencyjnym dla `k8s`. Projekt jest obecnie w fazie beta, w związku z tym nie będzie dalej eksplorowany.

2.3.1 Wybór narzędzia konfiguracyjnego

W części praktycznej zostały podjęte próby wykorzystania następujących konfiguratorów: `kubespary -cli`, `Rancher 2.0 Tech Preview 1 i 2`, `OpenShift Origin` oraz `kubespary`.

`OpenShift Origin` został odrzucony ze względu na brak modułów interfejsu graficznego.

`kubespary -cli` zostało odrzucone ze względu na brak kompatybilności z `kubespary`.

Uruchomienie Rancher 2.0 Tech Preview 1 zakończyło się sukcesem. Natomiast w trakcie pisania pracy pojawiła się kolejna wersja (Tech Preview 2), która tymczasowo utraciła wsparcie dla konfiguracji sprzętowego klastra i rozwiązanie zostało również odrzucone.

Testy praktyczne przeszło jedynie narzędzie `kubespary` i zostało wybrane do ostatecznej konfiguracji klastra na sieci wydziałowej.

2.4 Lista alternatyw Kubernetes

Wybór Kubernetes nie byłby w pełni uzasadniony bez krótkiej charakterystyki konkurencyjnych rozwiązań:

Fleet jest nakładką na `systemd` realizującą rozproszony system inicjalizacji systemów operacyjnych CoreOS. Kontenery są uruchamiane i zarządzane przez `systemd`, a stan przechowywany jest w `etcd`.

Obecnie projekt kończy swój żywot na rzecz `k8s` i w dniu 1 lutego 2018, został wycofany z domyślnej dystrybucji CoreOS. Nadal będzie dostępny w rejestrze pakietów CoreOS.

Docker Swarm jest rozwiązaniem zarządzania kontenerami zintegrowanym z systemem Docker. Główną zaletą jest niski próg wejścia i prostota, a wadą są małe możliwości w stosunku do innych rozwiązań.

Nomad jest narzędziem do generalnego zarządzania siecią komputerową, które oferuje między innymi 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 firmy HashiCorp, jak Consul i Vault.

Mesos jest najbardziej zaawansowanym i najefektywniej skalującym się rozwiązaniem zarządzania kontenerami. Jest również najbardziej skomplikowanym i trudnym w zarządzaniu rozwiązaniem. W związku z tym znajduje swoje zastosowanie tylko w największych sieciach komputerowych o zasięgu globalnym. # Praktyczny przegląd bezdyskowych systemów operacyjnych

Wszystkie moduły `k8s` są uruchamiane w kontenerach, więc dwoma podstawowymi wymaganiami systemu operacyjnego są możliwość uruchomienia Dockera i wsparcie wybranego narzędzia konfiguracyjnego. Dodatkowe wymagania związane z opisywanym w tej pracy przypadkiem użycia są:

- dostęp zdalny lub automatyczna konfiguracja dołączania do klastra `k8s`,
- wsparcie dla środowiska bezdyskowego,

Podstawowymi wyznacznikami przy wyborze systemu operacyjnego są:

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

Konfigurator `cloud-init`

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

`cloud-init`¹⁵ jest standardem oraz implementacją konfiguratora kompatybilnego z wieloma systemami operacyjnymi przeznaczonymi do działania w chmurze. Polega on na dostarczeniu pliku konfiguracyjnego w formacie `YAML`¹⁶ w trakcie lub tuż po rozruchu systemu operacyjnego.

Główną zaletą `cloud-init` jest tworzenie automatycznej i jednorodnej konfiguracji bazowych systemów operacyjnych w środowiskach chmurowych podobnie jak ma to miejsce na maszynach bezdyskowych.

`cloud-init` jest referencyjną implementacją standardu w Pythonie, co częściowo tłumaczy duży rozmiar obrazów przeznaczonych dla chmury. Po najmniejszych obrazach Pythona dla Dockera¹⁷ (`python:alpine` - 89MB i `python2:alpine` - 72 MB) wnioskuję, że nie istnieje mniejsza dystrybucja Pythona.

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

`coreos-cloudinit` jest częściową implementacją standardu w języku Go stworzoną na potrzeby systemu operacyjnego `coreos`. Rok temu projekt przestał być rozwijany¹⁸ i wychodzi z użytku.

`Rancher cloud-init` jest kontynuacją `coreos-cloudinit` dostosowaną do systemu operacyjnego `RancherOS`.

`clr-cloud-init` jest wewnętrzną implementacją standardu dla systemu `ClearLinux`. Powstał w celu optymalizacji standardu pod procesory firmy Intel i pozbycia się zależności od Pythona.

CoreOS

CoreOS¹⁹ jest pierwszą dystrybucją Linuxa przeznaczoną do zarządzania kontenerami. Charakteryzuje się kompletem narzędzi dedykowanych kontenerom oraz brakiem menadżera pakietów. Większość usług uruchamiana jest w nim przez `systemd` i bazuje na kontenerach.

Czysta instalacja zajmuje około 600 MB pamięci RAM i posiada najnowszą wersję Dockera oraz OverlayFS.

30 stycznia 2018 roku projekt został wykupiony przez Red Hat²⁰.

Obecnie CoreOS konfigurowany jest przez Container Linux Config²¹ konwertowany do formatu Ignition²². Minusem jest brak dystrybucji konwertera pod FreeBSD. Poprzednikiem Ignition był `coreos-cloudinit`.

RancherOS

RancherOS²³ jest systemem operacyjnym, w którym tradycyjny system inicjalizacji został zastąpiony trzema poziomami Dockera²⁴:

- `bootstrap_docker` - działa w initramie, czyli przygotowuje bazowy system,
- `system-docker` - zastępuje tradycyjny init oraz zarządza wszystkimi usługami systemowymi,
- `docker` - jest standardową dystrybucją Dockera, interakcja z nią 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. Nie jest często aktualizowany i posiada stare wersje zarówno Dockera (17.06 sprzed pół roku) jak i OverlayFS.

Narzędziem konfigurującym system jest przebudowany `coreos-cloudinit` → . Znaczną przewagą wobec oryginału jest możliwość sekwencyjnego uruchamiania dowolnej liczby plików konfiguracyjnych. Minimalna konfiguracja pozwalająca na zalogowanie się do systemu to:

```
#cloud-config
ssh_authorized_keys:
  - ssh-rsa <klucz RSA>
```

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

```
#!/bin/sh

cat << EOF > ssh.yml
#cloud-config
ssh_authorized_keys:
```

```
$(ssh-add -L | sed 's/^/ - /g')  
EOF
```

W związku z problemem w trakcie wczytywania konfiguracji²⁵ system operacyjny został odrzucony.

Project Atomic

Project Atomic²⁶ 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 Host²⁷,
- CentOS Atomic Host²⁸,
- Fedora Atomic Host²⁹.

Atomic Host są konfigurowane oficjalną implementacją cloud-inita.

Żadna z dystrybucji domyślnie nie wspiera rozruchu bezdyskowego, więc odrzucam te systemy operacyjne.

Alpine Linux

Alpine Linux³⁰ jest minimalną dystrybucją Linuxa bazowaną na musl-libc i busybox. System operacyjny wygląda obiecująco, ale próby rozruchu bezdyskowego zakończyły się niepowodzeniem i został odrzucony.

ClearLinux

ClearLinux³¹ 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 znacznie od innych dystrybucji. Znacznym minusem jest trudna w nawigowaniu dokumentacja systemu operacyjnego.

Wybór systemu operacyjnego

Spośród przetestowanych systemów wybrany został CoreOS. Czynniki decydującymi były:

- zawodność konfiguracji RancherOS,
- częstotliwość aktualizacji oprogramowania,
- kompletniejsza i łatwiejsza w nawigacji dokumentacja w porównaniu do ClearLinux.

Rozbieżność reszty parametrów była pomijalnie mała.

Rozdział 3

Praktyczne rozeznanie w narzędziach zarządzania klastrem Kubernetes

W tym rozdziale przeprowadzę czysto praktyczny przegląd wybranych rozwiązań na maszynach wirtualnych `VirtualBox` tak jak miało to miejsce w przeglądzie systemów operacyjnych.

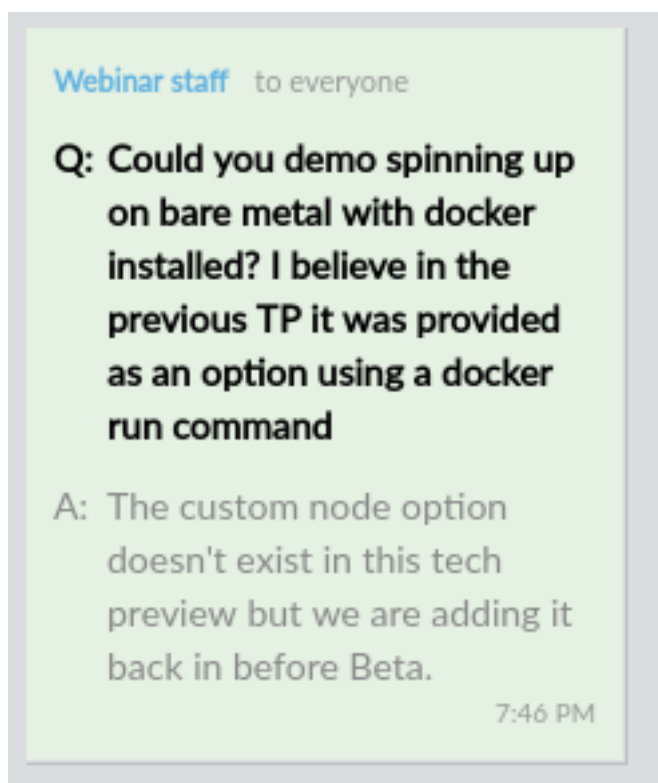
kubespray-cli

W trakcie testowania szybko okazało się, że narzędzie 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.

3.1 Rancher 2.0

Jest to wygodne narzędzie do uruchamiania i monitorowania klastra `k8s`. Wersja 2.0 (obecnie w fazie alpha) oferuje lepszą integrację z `k8s` 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 sprzętowej, więc jestem zmuszony odrzucić to rozwiązanie.



Uruchamianie wersji v2.0.0-alpha10

Testy przeprowadziłem na maszynach wirtualnych VirtualBox w trybie bezdyskowym systemu operacyjnego CoreOS z wykorzystaniem repozytorium ipxe-boot³².

Węzeł zarządzający uruchomiłem poniższą komendą:

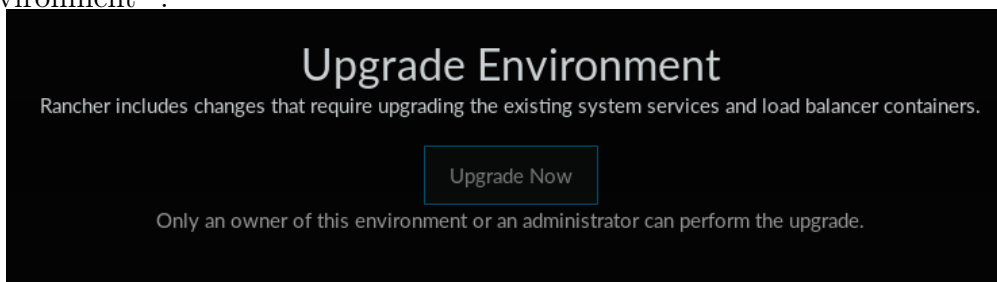
```
docker run --rm --name rancher -d -p 8080:8080 rancher/server:v2.0.0-  
↪ alpha10
```

Pierwszym krokiem było zalogowanie się do panelu administracyjnego Ranchera dostępnego na porcie 8080 lokalnej maszyny i uzyskanie komendy konfigurującej węzeł roboczy. Kolejnym i ostatnim krokiem konfiguracji było uruchomienie uzyskanej komendy na dowolnej liczbie węzłów roboczych:

```
sudo 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/B52944BEFAA613F0CE90  
↪ :1514678400000:E2yB6KfzxSix4YHti39BTw5RbKw
```

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

Napotkane błędy W wersji v2.0.0-alpha10 losowo pojawia się błąd Upgrade Environment³³.



Wnioski

Rancher na chwilę obecną (styczeń 2018 roku) jest bardzo wygodnym, ale również niestabilnym rozwiązaniem. Ze tego względu odrzucam Ranchera jako rozwiązanie problemu uruchamiania klastra k8s.

3.2 OpenShift Origin

Według dokumentacji³⁴ istnieją dwie metody uruchamiania serwera, w Dockerze i bezpośrednio na systemie Linux.

Zdecydowałem się uruchomić system w Dockerze ze względu na uniwersalność rozwiązania. Komendą uruchamiającą serwer był:

```
docker run -d --name "origin" \
  --privileged --pid=host --net=host \
  -v /:/rootfs:ro \
  -v /var/run:/var/run:rw \
  -v /sys:/sys \
  -v /sys/fs/cgroup:/sys/fs/cgroup:rw \
  -v /var/lib/docker:/var/lib/docker:rw \
  -v /var/lib/origin/openshift.local.volumes:/var/lib/origin/openshift.
  ↪ local.volumes:rslave \
  openshift/origin start --public-master
```

W porównaniu do przykładu z dokumentacji dodany został argument --
↪ public-master w celu uruchomienia interfejsu graficznego.

Sterowniki cgroup

Większość dystrybucji Linuxa (np. Arch, CoreOS, Fedora, Debian) nie konfiguruje sterownika cgroup Dockera i korzysta z domyślnego cgroupfs, wersję sterownika można zweryfikować komendą `docker info | grep -i cgroup`.

OpenShift korzysta ze sterownika systemd, a kubelet w trakcie rozruchu weryfikuje zgodność sterowników, co skutkuje niekompatybilnością z domyślną konfiguracją Dockera³⁵, objawiającą się poniższym błędem:

```
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 dodając argumenty `--exec-opt native.cgroupdriver=systemd` do komendy uruchamiającej główny proces Dockera.

Próba uruchomienia serwera na systemie Arch Linux

Po rozwiązaniu powyższego problemu i udanym uruchomieniu ww. komendy rozruchowej przeszedłem do dalszych etapów instalacji zgodnie z dokumentacją:

1. Uruchomiłem linię komend OpenShift i zalogowałem się jako użytkownik testowy:

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

```
$ oc login
Username: test
Password: test
```

2. Stworzyłem nowy projekt:

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

3. Pobrałem przykładową aplikację:

```
$ oc tag --source=docker openshift/deployment-example:v1 deployment-
  ↳ example:latest
```

4. Uruchomiłem instancję aplikacji:

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

Uruchomienie przykładowej zakończyło się poniższym błędem:

```
$ watch -n 5 oc status
In project test on server https://192.168.0.87:8443

svc/deployment-example - 172.30.52.184:8080
  dc/deployment-example deploys istag/deployment-example:latest
  deployment #1 failed 1 minute ago: config change
```

Próba uruchomienia OpenShift Origin na Arch Linux zakończyła się niepowodzeniem.

Próba uruchomienia serwera na Fedora Atomic Host w VirtualBox

Następnie spróbowałem uruchomić OpenShift Origin na oficjalnie wspieranym systemie operacyjnym Fedora Atomic Host z użyciem Vagranta w poniższej konfiguracji Vagrantfile:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  config.vm.box = "fedora/27-atomic-host"
  config.vm.box_check_update = false
  config.vm.network "forwarded_port", guest: 8443, host: 18443, host_ip:
    ↪ "127.0.0.1"
  config.vm.network "forwarded_port", guest: 8080, host: 18080, host_ip:
    ↪ "127.0.0.1"
  config.vm.provider "virtualbox" do |vb|
    vb.gui = false
    vb.memory = "8192"
  end
  config.vm.provision "shell", inline: <<-SHELL
  SHELL
end
```

Maszynę z lokalnym dyskiem uruchomiłem komendami `vagrant up`, a następnie uzyskałem do niej dostęp komendą `vagrant ssh`. Kroki 1-4 były analogiczne do uruchamiania na Arch Linux, po czym przeszedłem do kolejnych kroków z dokumentacji:

5. Oczekałem aż przykładowa aplikacja się uruchomi i zweryfikowałem jej działanie:

```
$ watch -n 5 oc status
In project test on server https://10.0.2.15:8443
```



```
svc/deployment-example - 172.30.221.105:8080
dc/deployment-example deploys istag/deployment-example:latest
deployment #1 deployed 3 seconds ago - 1 pod
```

```
$ curl http://172.30.221.105:8080 | grep v1
<div class="box"><h1>v1</h1><h2></h2></div>
```

6. Przeprowadziłem aktualizację aplikacji i ponownie zweryfikowałem jej działanie:

```
$ oc tag --source=docker openshift/deployment-example:v2 deployment-
↪ example:latest
Tag deployment-example:latest set to openshift/deployment-example:v2.
```

```
$ watch -n 5 oc status
In project test on server https://10.0.2.15:8443

svc/deployment-example - 172.30.221.105:8080
dc/deployment-example deploys istag/deployment-example:latest
deployment #2 running for 8 seconds - 1 pod
deployment #1 deployed 8 minutes ago - 1 pod
```

```
$ curl -s http://172.30.221.105:8080 | grep v2
<div class="box"><h1>v2</h1><h2></h2></div>
```

7. Próba dostania się do graficznego panelu administracyjnego zakończyła się niepowodzeniem:

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

Nie znalazłem rozwiązania powyższego problemu w internecie, ani na oficjalnym kanale #openshift na irc.freenode.net.

Wnioski

Panel administracyjny klastra OpenShift Origin jest jedyną znaczącą przewagą nad kubespray. Reszta zarządzania klastrem odbywa się również za pomocą repozytorium skryptów Ansibla (w tym dodawanie kolejnych węzłów klastra³⁶).

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

3.3 kubespray

Kroki podjęte w celu przetestowania narzędzia `kubespray` zostały zawarte w skrypcie `bin/setup-cluster` z repozytorium `kubernetes-cluster`³⁷. Pełny kod źródłowy został zawarty w “Dodatku B”.

W trakcie testowania narzędzia `kubespray` nie napotkałem żadnych problemów, a szczegóły konfiguracji zawarłem jedynie w kolejnych rozdziałach.

Rozdział 4

Wstępna implementacja kubespray w laboratorium 225

Po wybraniu narzędzia `kubespray` przystąpiłem do jego uruchomienia w laboratorium 225 w celu dopracowania ostatecznej konfiguracji.

Repozytorium `kubernetes-cluster` pobrałem do folderu `/home/stud/nazarewk`
→ `/kubernetes/kubernetes-cluster` na maszynie `ldap`.

Konieczne było przygotowanie dwóch plików konfiguracyjnych rozruch CoreOS.

`zetis/WWW/boot/coreos.ipxe` jest plikiem konfiguracyjnym programu rozruchowego iPXE. Został udostępniony pod adresem `http://vol/~nazarewk`
→ `/boot/coreos.ipxe`, zawierał:

```
#!ipxe
set http http://vol
set ftp http://ftp/pub/

set base-url ${ftp}/Linux/CoreOS/alpha
set ignition ${http}/~nazarewk/boot/coreos.ign

set opts ${opts} coreos.autologin
set opts ${opts} coreos.first_boot=1 coreos.config.url=${ignition}
set opts ${opts} systemd.journald.max_level_console=debug
kernel ${base-url}/coreos_production_pxe.vmlinuz ${opts}
initrd ${base-url}/coreos_production_pxe_image.cpio.gz

boot
```

`zetis/WWW/boot/coreos.yml` jest plikiem Container Linux Config, z którego skryptem `bin/render-coreos` generowany jest docelowy plik konfiguracyjny Ignition `zetis/WWW/boot/coreos.ign`.

Przykładem pliku `coreos.yml` tworzącego użytkownika `admin` w grupach `docker` i `sudo` oraz dodającego klucz dostępowy SSH jest:

```
passwd:
  users:
  - name: admin
    groups: [sudo, docker]
    ssh_authorized_keys:
    - ssh-rsa <klucz RSA> nazarewk
```

Na podstawie powyższego pliku skrypt `bin/render-coreos` generuje następujący plik `coreos.ign` w formacie JSON:

```
{
  "ignition": {
    "config": {},
    "timeouts": {},
    "version": "2.1.0"
  },
  "networkd": {},
  "passwd": {
    "users": [
      {
        "groups": [
          "sudo",
          "docker"
        ],
        "name": "admin",
        "sshAuthorizedKeys": [
          "ssh-rsa <klucz RSA> nazarewk"
        ]
      }
    ]
  },
  "storage": {},
  "systemd": {}
}
```

Plik `coreos.ign` został udostępniony pod adresem <http://vol/~nazarewk/boot/coreos.ign>.

Po przygotowaniu powyższych plików konieczne było wskazanie ich maszynom wywołując poniższą komendę na maszynie `volt`:

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

4.1 Konfiguracja kubespary

Konfiguracja `kubespary` ogranicza się do skopiowania przykładowego inwentarza i edycji plików `inventory.cfg` oraz `group_vars/all.yml`.

inventory.cfg jest plikiem INI reprezentującym topologię klastra k8s. Składa się on z następujących sekcji:

- definicji wszystkich węzłów klastra w sekcji `[all]`,
- węzłów na których uruchomiony jest klaster etcd w sekcji `[etcd]`,
- węzłów zarządzających w sekcji `[kube-master]`,
- węzłów roboczych w sekcji `[kube-node]`,
- połączenia sekcji `[kube-master]` i `[kube-node]` w sekcję `[k8s-cluster]`.

Kompletnym przykładem pliku dla laboratorium 225 jest:

```
[all]
s4  ansible_host=s4 ip=10.146.225.4 ansible_python_interpreter=/opt/bin/
    ↪ python
s5  ansible_host=s5 ip=10.146.225.5 ansible_python_interpreter=/opt/bin/
    ↪ python
s6  ansible_host=s6 ip=10.146.225.6 ansible_python_interpreter=/opt/bin/
    ↪ python
;s8  ansible_host=s8 ip=10.146.225.8 ansible_python_interpreter=/opt/bin
    ↪ /python
;s9  ansible_host=s9 ip=10.146.225.9 ansible_python_interpreter=/opt/bin
    ↪ /python

[kube-master]
s4

[kube-node]
s5
s6

[etcd]
s4

[k8s-cluster:children]
kube-node
kube-master
```

group_vars/all.yml jest plikiem YAML zawierającym parametry wywołania kubespary wspólne dla wszystkich węzłów.

Minimalną konfiguracją dla systemu CoreOS jest:

```
bootstrap_os: coreos # automatyczna instalacja Pythona na CoreOS
kube_basic_auth: true # podstawowe logowanie do Dashboardu Kubernetes
kubeconfig_localhost: true # konfiguracja lokalnego kubectl
download_run_once: true # pozwala na jednorazowe pobieranie plików
```

4.2 Napotkane problemy

Klonowanie repozytoriów bez logowania

W celu umożliwienia anonimowego klonowania repozytorium z Githuba, konieczne było wykorzystanie protokołu `https` zamiast `git`:

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

Zmiana musiała być wprowadzona również do submodułów `git`.

Zdejmowanie atrybutu wykonywalności pliku

W konfiguracji uczelnianej `git` nie ustawia i zdejmuje atrybut wykonalności pobieranych plików. Po każdej komendzie `git pull` konieczne było wywołanie komendy `chmod +x bin/*`.

Konfiguracja bezhasłowego dostępu SSH

W celu uruchomienia narzędzia `kubespray` w trybie bezinterakcyjnym konieczna była konfiguracja bezhasłowego dostępu SSH do zarządzanych maszyn. Powyższy problem rozwiązuje dodanie poniższych wpisów w pliku `~/.ssh/config`:

→ `ssh/config`:

```
Host s?  
  User admin  
  StrictHostKeyChecking no  
  UserKnownHostsFile /dev/null  
  IdentityFile ~/.ssh/id_rsa  
  IdentitiesOnly yes
```

Problemy z siecią uczelnianą

W trakcie pierwszego uruchamiania `kubespray` występowały przeciążenia sieci uczelnianej, w związku z tym komendy były tracone. Problemowi zaradzono konfigurując Ansible do powtarzania nieudanych komend.

Ograniczenie 3 serwerów DNS w kubespray

`kubespray` nie radzi sobie z więcej niż trzema wpisami DNS na konfigurowanych maszynach. Ograniczenie to objawiło się na maszynie `s8`, która konfigurowała dodatkowy interfejs sieciowy.

Problemu można uniknąć wyłączając zbędne interfejsy sieciowe.

Niewystarczająca ilość pamięci RAM na maszynie s2

Maszyna s2 nie posiada ilości pamięci RAM wystarczającej do uruchomienia węzła k8s i nie powinna być wykorzystywana.

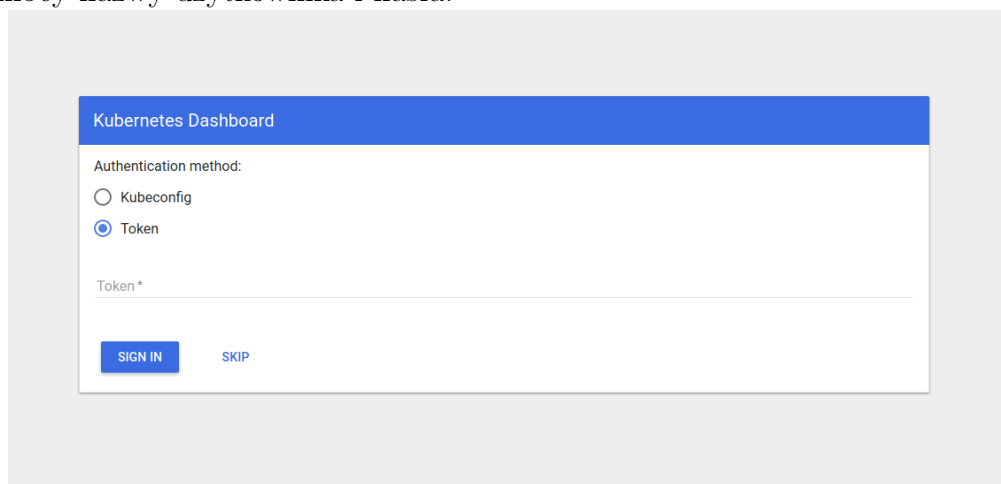
Eksperymentalna konfiguracja narzędzia Vault w kubespray

W trakcie testów `kubespray` zostało rozszerzone o wsparcie narzędzia `Vault` firmy HashiCorp. Rozszerzenie okazało się niekompletne i konieczne było jego wyłączenie.

Logowanie do Kubernetes Dashboard

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

W starszej wersji k8s i/lub `kubespray` brakowało opcji logowania przy pomocy nazwy użytkownika i hasła:



Od 29 stycznia 2018 roku zaobserwowany został poprawny ekran logowania (opcja `Basic`):

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` →). Ze względu na liczne pojawiające się błędy temat został zarzucony.

Uruchomienie Helm

Instalacja menadżera pakietów sprowadza się do:

1. ściągnięcia pliku wykonywalnego `helm`,
2. stworzenie roli RBAC dla serwerowego komponentu `Tiller`,
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 jest uruchamiana przez `SSH` na węźle zarządzającym (domyślnie `s4`).

Większość dostępnych pakietów wymaga trwałych zasobów dyskowych, więc temat nie był dalej zgłębiany.

Rozdział 5

Prezentacja docelowej konfiguracji klastra Kubernetes w sieci uczelnianej

W tym rozdziale opisana jest pełna procedura uruchomienia klastra Kubernetes ↪ z wykorzystaniem opracowanych skryptów. Wywoływane komendy są w formacie `< nazwa_hosta >% <komenda>`. Kolejnymi liniami fragmentów kodu jest standardowe wyjście wywoływanych komend, a trzy kropki ... reprezentują jego wycięte fragmenty.

Pełną konfiguracja k8s można uruchomić z maszyny 1dap; znajduje się ona w folderze `/pub/Linux/CoreOS/zetis/kubernetes`, który zawiera następujące pliki i foldery:

- `kubernetes-cluster` - moje repozytorium zawierające konfigurację i skrypty pozwalające uruchomić klaster,
- `boot` - skrót do folderu `kubernetes-cluster/zetis/www/boot` zawierającego konfigurację iPXE oraz Ignition:
 - `coreos.ipxe` - konfiguracja iPXE,
 - `coreos.ign` - konfiguracja Ignition,

5.1 Procedura uruchomienia klastra

1. Wejście na maszynę 1dap do folderu `/pub/Linux/CoreOS/zetis/kubernetes` ↪ `/kubernetes-cluster`,
2. Upewnienie się, że klucz dostępowy SSH jest obecny w `boot/coreos.ign`,
3. Uruchomienie maszyn-węzłów wybierając z menu iPXE CoreOS -> k8s lub wybierając w narzędziu boot bezpośrednio `coreos kub`,

4. Weryfikacja bezhasłowego dostępu do węzłów, minimalna konfiguracja
~/.ssh/config to:

```
Host s?  
  User admin  
  IdentityFile ~/.ssh/id_rsa  
  IdentitiesOnly yes  
  StrictHostKeyChecking no  
  UserKnownHostsFile /dev/null
```

5. W razie braku, utworzenie folderu `kubespray / my_inventory` komendą `cp`
→ `-rav kubespray / inventory kubespray / my_inventory`
6. Upewnienie się, że uruchomione maszyny są obecne w odpowiednich sekcjach pliku `inventory / inventory .cfg`, domyślna konfiguracja pozwala na uruchomienie klastra na maszynach `s4`, `s5` i `s6` z jednym węzłem zarządzającym `s4`:

```
[all]  
;s3 ip=10.146.225.3  
s4 ip=10.146.225.4  
s5 ip=10.146.225.5  
s6 ip=10.146.225.6  
;s7 ip=10.146.225.7  
;s8 ip=10.146.225.8  
;s9 ip=10.146.225.9  
;sa ip=10.146.225.10  
;sb ip=10.146.225.11  
;sc ip=10.146.225.12  
  
[kube-master]  
s4  
  
[kube-node]  
s5  
s6  
  
[etcd]  
s4  
  
[k8s-cluster:children]  
kube-node  
kube-master
```

7. Upewnienie się, że plik `inventory / group_vars / all .yaml` zawiera poprawną konfigurację, na przykład:

```
cluster_name: zetis-kubernetes
bootstrap_os: coreos
kube_basic_auth: true
kubeconfig_localhost: true
kubectl_localhost: true
download_run_once: true
```

8. Konfiguracja klastra skryptem `bin/setup-cluster`. Po około 10 minutach skrypt powinien zakończyć się wpisami pokroju:

```
...
PLAY RECAP *****
localhost                : ok=2    changed=0    unreachable=0
  ↪ failed=0
s4                        : ok=281  changed=94   unreachable=0
  ↪ failed=0
s5                        : ok=346  changed=80   unreachable=0
  ↪ failed=0
s6                        : ok=186  changed=54   unreachable=0
  ↪ failed=0
...
```

9. Weryfikacja konfiguracji węzłów:

```
ldap% bin/kubectl get nodes
NAME      STATUS    ROLES    AGE      VERSION
s4        Ready     master   2m       v1.9.1_coreos.0
s5        Ready     node     2m       v1.9.1_coreos.0
s6        Ready     node     2m       v1.9.1_coreos.0
```

5.2 Procedury korzystania z klastra Kubernetes

Aby dodać użytkownika wywołujemy skrypt `bin/students` z pierwszym argumentem będącym nazwą użytkownika i drugim `create`.

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

```
apiVersion: v1
kind: Namespace
metadata:
  name: nazarewk
  labels:
```

```

    name: nazarewk
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nazarewk
  namespace: nazarewk
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nazarewk-admin-binding
  namespace: nazarewk
roleRef:
  kind: ClusterRole
  name: admin
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: nazarewk

```

Tworzone są następujące obiekty Kubernetes:

- przestrzeń nazw użytkownika,
- konto serwisowe użytkownika,
- przypisanie konta serwisowemu roli `admin`, która daje pełny dostęp do jego przestrzeni nazw.

Przykładowa procedura korzystania z klastra

1. Stworzenie konta użytkownika

```

ldap% bin/students nazarewk create
namespace "nazarewk" created
serviceaccount "nazarewk" created
rolebinding "nazarewk-admin-binding" created
Tokens:
eyJhbGciOiJSUzI1NiIsImtpdiI6ImlzbnVpdCIsImVhbnQiOiJhHfxU-TRw

```

```

ldap% bin/students

```

NAME	STATUS	AGE
default	Active	3m
kube-public	Active	3m
kube-system	Active	3m
nazarewk	Active	16s

2. Skopiowanie przepustki użytkownika na (przykładową) maszynę s2 z działającym systemem operacyjnym Ubuntu:

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

3. Pobranie narzędzia kubectl:

```
s2% cd /tmp
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
s2% chmod +x kubectl
s2% sudo mv kubectl /usr/local/bin
s2% source <(kubectl completion zsh)
```

4. Weryfikacja braku konfiguracji narzędzia:

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

5. Konfiguracja kubectl w najprostszy możliwy sposób, czyli aliasem:

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

6. Weryfikacja braku dostępu do zasobów globalnych klastra:

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

7. Stworzenie Deployment przykładowej aplikacji:

```
s2% kubectl run echoserver --image=gcr.io/google_containers/echoserver
    ↪ :1.4 --port=8080 --replicas=2
deployment "echoserver" created
```

```
s2% kubectl get deployments
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
echoserver     2         2         2            2           3m
```

```
s2% kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
echoserver-7b9bbf6ff-22df4	1/1	Running	0	4m
echoserver-7b9bbf6ff-c6kbv	1/1	Running	0	4m

8. Stworzenie serwisu typu `NodePort` w celu dostępu do aplikacji:

```
s2% kubectl expose deployment echoserver --type=NodePort
service "echoserver" exposed
```

```
s2% kubectl describe services/echoserver | grep -e NodePort:
NodePort:                <unset>    30100/TCP
```

Wynik ostatniej komendy oznacza dostęp do aplikacji przez port 30100 dowolnego węzła klastra.

9. Weryfikacja działania aplikacji:

```
s2% curl s4:30100
CLIENT VALUES:
client_address=10.233.107.64
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://s4:8080/

SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=/*/*
host=s4:30100
user-agent=curl/7.47.0
BODY:
-no body in request-
```

10. Weryfikacja dostępu do aplikacji z maszyny `ldap`:

```
ldap% curl s4:30100
CLIENT VALUES:
client_address=10.233.107.64
command=GET
real path=/
query=nil
request_version=1.1
request_uri=http://s4:8080/
```

```
SERVER VALUES:
server_version=nginx: 1.10.0 - lua: 10001

HEADERS RECEIVED:
accept=/*/*
host=s4:30100
user-agent=curl/7.58.0
BODY:
-no body in request-
```

11. Usunięcie użytkownika:

```
ldap% bin/students nazarewk delete
namespace "nazarewk" deleted
serviceaccount "nazarewk" deleted
rolebinding "nazarewk-admin-binding" deleted
Tokens:
Error from server (NotFound): serviceaccounts "nazarewk" not found
```

12. Weryfikacja usunięcia wszystkich zasobów użytkownika:

```
ldap% curl s4:30100
curl: (7) Failed to connect to s4 port 30100: Connection refused
```

```
ldap% bin/kubectl get namespace
NAME          STATUS    AGE
default       Active    46m
kube-public   Active    46m
kube-system   Active    46m
```


Rozdział 6

Podsumowanie

W niniejszej pracy omówiono zagadnienie konfiguracji klastra `Kubernetes`. Przedstawione zostały elementarne pojęcia z zakresu konfiguracji i korzystania z klastra `k8s` bazowanego na maszynach bezdyskowych. Bezpośrednim rezultatem pracy są skrypty implementujące klaster w sieci uczelnianej oraz umożliwiające prowadzenie zajęć demonstracyjnych ze studentami.

W związku z aktywnym rozwojem projektu `k8s` stworzenie publikacji wyczerpującej temat nie jest możliwe. Wiele obiecujących narzędzi pojawiało się i znacznie zmieniał w trakcie pisania niniejszej pracy. Nie zostały one należycie opisane lub nie nadawały się jeszcze do użytku w momencie jej zakończenia. Projekty te mogłyby realizować alternatywne implementacje poruszonych w tej pracy zagadnień.

Ze względu na ograniczenia czasowe praca nie poruszyła zaawansowanych zagadnień `k8s` oraz nie zaimplementowała żadnych mechanizmów bazujących na zewnętrznych usługach. Przykładami zagadnień, które mogłyby zostać w przyszłości opisane, są:

- integracja z sieciowymi systemami plików w celu utrwalenia danych,
- bezobsługowe dołączanie nowych węzłów do klastra `k8s` działającego w sieci uczelnianej,
- opracowanie zajęć laboratoryjnych bazowanych na `Kubernetes`.

Przedstawiony we wstępie cel został osiągnięty, a wysiłek włożony w realizację pracy zaowocował znacznym zwiększeniem kompetencji autora związanych z tematyką konteneryzacji i `Kubernetes`. Zdobyta wiedza jest bezcenną wartością na rynku pracy.

Dodatek A

Wykaz odnośników

- 1 <https://www.opencontainers.org/about>
- 2 <https://kubernetes.io/>
- 3 <https://github.com/coreos/flannel#flannel>
- 4 <https://www.projectcalico.org/>
- 5 <https://github.com/kubernetes/ingress-nginx>
- 6 <https://github.com/kubernetes/ingress-gce>
- 7 <https://kubernetes.io/docs/admin/authorization/rbac/>
- 8 <https://github.com/kubernetes/community/blob/master/contributors/devel/api-conventions.md>
- 9 <https://en.wikipedia.org/wiki/Cron>
- 10 <https://github.com/kubernetes/charts>
- 11 <https://maas.io/>
- 12 <https://typhoon.psdn.io/bare-metal/>
- 13 <https://github.com/openshift/origin>
- 14 <https://mesosphere.com/blog/kubernetes-dcos/>
- 15 <https://cloud-init.io/>
- 16 <http://yaml.org/>
- 17 https://hub.docker.com/_/python/
- 18 <https://github.com/coreos/coreos-cloudinit/commit/3460ca4414fd91de66cd581d997bf453fd895b6>
- 19 <https://coreos.com/>
- 20 <https://www.redhat.com/en/about/press-releases/red-hat-acquire-coreos-expanding-its-kuber>
- 21 <https://coreos.com/os/docs/latest/provisioning.html>
- 22 <https://coreos.com/ignition/docs/latest/>
- 23 <https://rancher.com/rancher-os/>
- 24 <http://rancher.com/docs/os/latest/en/configuration/docker/>
- 25 <https://github.com/rancher/os/issues/2204>
- 26 <https://www.projectatomic.io/>

27 <https://www.redhat.com/en/resources/enterprise-linux-atomic-host-datasheet>
28 <https://wiki.centos.org/SpecialInterestGroup/Atomic/Download/>
29 <https://getfedora.org/atomic/download/>
30 <https://alpinelinux.org/>
31 <https://clearlinux.org/>
32 <https://github.com/nazarewk/ipxe-boot>
33 <https://github.com/rancher/rancher/issues/10396>
34 https://docs.openshift.org/latest/getting_started/administrators.html
35 <https://github.com/openshift/origin/issues/14766>
36 https://docs.openshift.com/enterprise/3.0/admin_guide/manage_nodes.html#adding-nodes
37 <https://github.com/nazarewk/kubernetes-cluster>

Dodatek B

Wykaz kodu źródłowego

B.1 repozytorium kubernetes-cluster

kubernetes-cluster (<https://github.com/nazarewk/kubernetes-cluster>) jest repozytorium Git, które zawiera kompletny kod źródłowy wykorzystywany w tej pracy inżynierskiej.

Kod można znaleźć i uruchomić w sieci uczelnianej z katalogu /pub/Linux
↔ /CoreOS/zetis/kubernetes/kubernetes-cluster/ na maszynie ldap.

bin/pull

Pobiera aktualną wersję repozytorium wraz z najnowszą wersją zależności:

```
#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: git pull wrapper including submodule updates

git pull || (git fetch --all && git reset --hard origin)
git submodule update --init --remote --recursive
chmod +x ${0%/*}/*
```

bin/vars

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

```
#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: Holds common variables used in all other scripts

if [ -z "$KC_CONFIGURED" ]; then
    KC_CONFIGURED=1
```

```

BIN="$(realpath ${0%/*})"
PATH="$BIN:$PATH"
PDIR="${BIN%/*}"
VENV="$PDIR/.env"
KUBESPRAY="$PDIR/kubespray"
INVENTORY="$KUBESPRAY/my_inventory"
CONFIG_FILE="$INVENTORY/inventory.cfg"

export KUBECONFIG="$KUBESPRAY/artifacts/admin.conf"
export ANSIBLE_CONFIG="$PDIR/ansible.cfg"
fi

```

bin/ensure-virtualenv

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

```

#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: installs python virtualenv

. ${0%/*}/vars
python="$(which python)"
activate="$VENV/bin/activate"

find_virtualenv() {
    which virtualenv 2> /dev/null || which virtualenv
}

install_virtualenv () {
    # Installs pip and/or virtualenv

    local pip=$(which pip)
    local apt=$(which apt)
    if [ ! -z "$pip" ]; then
        sudo $pip install virtualenv
        return 0
    elif [ ! -z "$apt" ]; then
        sudo $apt install virtualenv
        return 0
    else
        echo 'Virtualenv, pip and apt-get are both missing, cannot proceed'
        exit 1
    fi
}

create_env () {
    local cmd=$(find_virtualenv)
    [ -z "$cmd" ] && install_virtualenv && cmd=$(find_virtualenv)

    $cmd -p $python $VENV
}

```

```

    . $activate

    pip install -U pip setuptools
}

[ -z "$python" ] && echo 'python is missing' && exit 1
[ -e "$activate" ] && . $activate || create_env
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:

```

#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: configures cluster (SSH, inventory, group_vars)

. ${0%*/}/vars
SSH_CFG_DST="$HOME/.ssh/config"
SSH_CFG="$PDIR/zetis/.ssh/kubernetes.conf"

is_ssh_config_missing () {
    cat << EOF | python
import sys
expected = open('$SSH_CFG', 'rb').read()
existing = open('$SSH_CFG_DST', 'rb').read()
sys.exit(expected in existing)
EOF
    return $?
}

is_ssh_config_missing && cat $SSH_CFG >> $SSH_CFG_DST

file="$INVENTORY/inventory.cfg"
[ -f "$file" ] || cat << EOF > "$file"
[all]
;s3 ip=10.146.225.3
s4 ip=10.146.225.4
s5 ip=10.146.225.5
s6 ip=10.146.225.6
;s7 ip=10.146.225.7
;s8 ip=10.146.225.8
;s9 ip=10.146.225.9
;sa ip=10.146.225.10
;sb ip=10.146.225.11
;sc ip=10.146.225.12

[kube-master]
s4

```

```
[kube-node]
s5
s6

[etcd]
s4

[k8s-cluster:children]
kube-node
kube-master
EOF

file="$INVENTORY/group_vars/all.yml"
[ -f "$file" ] || cat << EOF > "$file"
cluster_name: zetis-kubernetes
bootstrap_os: coreos
kube_basic_auth: true
kubeconfig_localhost: true
kubectl_localhost: true
download_run_once: true
EOF
```

bin/render-coreos

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

```
#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: renders CoreOS Ignition (incl. retrieving binary)

. ${0%/*}/vars
ct_version=${1:-0.6.1}
boot="$PDIR/zetis/WWW/boot"
url='https://github.com/coreos/container-linux-config-transpiler'
url="$url/releases/download"
url="$url/v${ct_version}/ct-v${ct_version}-x86_64-unknown-linux-gnu"

wget -nc $url -O $BIN/ct
chmod +x $BIN/ct
$BIN/ct -pretty \
  -in-file "$boot/coreos.yml" \
  -out-file "$boot/coreos.ign"
```

bin/setup-cluster

Właściwy skrypt konfiguruje klastery na działających maszynach CoreOS:

```
#!/bin/sh
```

```

# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: sets up the cluster

. ${0%/*}/vars
. $BIN/ensure-configuration
. $BIN/activate

(
    cd $KUBESPRAY;
    ansible-playbook -i $INVENTORY/inventory.cfg cluster.yml -b -v $@
)

chmod -R 700 $KUBESPRAY/artifacts/admin.conf $KUBESPRAY/credentials

cat << EOF
Login credentials:
    user: kube
    password: $(cat $PDIR/credentials/kube_user)
EOF

```

bin/setup-cluster-full

Skrót do pobierania najnowszej wersji, a następnie uruchamiania klastra:

```

#!/bin/sh
. ${0%/*}/vars

$BIN/pull
$BIN/setup-cluster $@

```

bin/setup-cluster-upgrade

Skrypt analogiczny do `setup-cluster`, ale wywołujący `upgrade-cluster`.

→ yml zamiast `cluster.yml`:

```

#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: runs upgrade-cluster.yml instead of cluster.yml

. ${0%/*}/vars
. $BIN/ensure-configuration
. $BIN/activate

cd $KUBESPRAY
ansible-playbook -i $INVENTORY/inventory.cfg upgrade-cluster.yml -b -v
→ $@

```

bin/kubect1

Skrót `kubect1` z automatycznym dołączaniem konfiguracji `kubespray`:


```
#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: kubectl wrapper which uses generated configurations

# Handle running kubectl from PATH
[ "${0%/*}" == "$0" ] \
  && _bin=$(dirname $(which kubectl)) \
  || _bin=${0%/*}

. $_bin/vars

local_bin="$PDIR/artifacts/kubectl"

if [ -f $local_bin ]; then
  chmod +x $local_bin
  kubectl=$local_bin
else
  kubectl=/usr/bin/kubectl
fi

$kubectl $@
```

bin/students

Skrypt zarządzający obiektami k8s użytkowników, czyli studentów:

```
#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: executes operations on users (create, get, describe, delete)

. ${0%/*}/vars

kubectl_command () {
  local users="$1"
  local cmd="$2"
  for name in $users; do
    cat <<EOF | $BIN/kubectl $cmd -f -
apiVersion: v1
kind: Namespace
metadata:
  name: ${name}
  labels:
    name: ${name}

---

apiVersion: v1
kind: ServiceAccount
metadata:
  name: ${name}
  namespace: ${name}
```

```

---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: ${name}-admin-binding
  namespace: ${name}
roleRef:
  kind: ClusterRole
  name: admin
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: ${name}
EOF
echo Tokens:
$BIN/student-tokens ${name}
done
}

case $1 in
  ""|list)
    $BIN/kubect1 get namespace
    ;;
  -h)
    cat << EOF
    Usage: $0 "<usernames>" create|get|describe|delete
    where <usernames> is a single-argument list of usernames
    You can also call it without arguments to list users (namespaces)
    EOF
    ;;
  *)
    kubect1_command "$@"
    ;;
esac

```

bin/student-tokens

Listuje przepustki konkretnego użytkownika:

```

#!/bin/sh
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>
# Purpose: lists kubernetes authentication tokens for specified user

. ${0%/*}/vars
name=$1
args="--namespace $name get serviceaccount $name"
args="$args -o jsonpath={.secrets[*].name}"
for token in ` $BIN/kubect1 $args `; do
  args="get secrets $token --namespace $name"
  args="$args -o jsonpath={.data.token}"
done

```

```
$BIN/kubectl $args | base64 --decode  
echo  
done
```

bin/install-helm

Instaluje menadżer pakietów Helm:

```
#!/bin/sh  
# Author: Krzysztof Nazarewski <nazarewk@gmail.com>  
# Purpose: installs helm package manager into Kubernetes cluster  
  
. ${0%/*}/vars  
  
host=${1:-s4}  
  
ssh () {  
    command ssh $host sudo $@  
}  
  
cat <<EOF | ssh bash  
export HELM_INSTALL_DIR=/opt/bin  
url='https://raw.githubusercontent.com/kubernetes/helm/master/scripts/  
    ↪ get'  
[ -f /opt/bin/helm ] || curl \"$url\" | bash  
EOF  
  
cat <<EOF | ssh kubectl create -f -  
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  name: tiller  
  namespace: kube-system  
---  
apiVersion: rbac.authorization.k8s.io/v1beta1  
kind: ClusterRoleBinding  
metadata:  
  name: tiller  
roleRef:  
  apiGroup: rbac.authorization.k8s.io  
  kind: ClusterRole  
  name: cluster-admin  
subjects:  
- kind: ServiceAccount  
  name: tiller  
  namespace: kube-system  
EOF  
  
ssh helm version | grep Server: || ssh helm init --service-account  
    ↪ tiller
```

zetis/.ssh/kubernetes.conf

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

```
Host s?  
  User admin  
  IdentityFile ~/.ssh/id_rsa  
  IdentitiesOnly yes  
  StrictHostKeyChecking no  
  UserKnownHostsFile /dev/null
```

zetis/WWW/boot/coreos.ipxe

Skrypt iPXE uruchamiający maszynę z CoreOS:

```
#!/ipxe  
set ftp http://ftp/pub/  
  
set base-url ${ftp}/Linux/CoreOS/alpha  
set ignition ${ftp}/Linux/CoreOS/zetis/kubernetes/boot/coreos.ign  
  
set opts ${opts} coreos.autologin  
set opts ${opts} coreos.first_boot=1 coreos.config.url=${ignition}  
set opts ${opts} systemd.journald.max_level_console=debug  
kernel ${base-url}/coreos_production_pxe.vmlinuz ${opts}  
initrd ${base-url}/coreos_production_pxe_image.cpio.gz  
  
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:

```
passwd:  
  users:  
    - name: admin  
      groups: [sudo, docker]  
      ssh_authorized_keys:  
        - ssh-rsa <klucz RSA> ato@volt.iem.pw.edu.pl  
        - ssh-rsa <klucz RSA> nazarewk  
        - ssh-rsa <klucz RSA> nazarewk@ldap.iem.pw.edu.pl
```

zetis/WWW/boot/coreos.ign

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

```

{
  "ignition": {
    "config": {},
    "timeouts": {},
    "version": "2.1.0"
  },
  "networkd": {},
  "passwd": {
    "users": [
      {
        "groups": [
          "sudo",
          "docker"
        ],
        "name": "admin",
        "sshAuthorizedKeys": [
          "ssh-rsa <klucz RSA> nazarewk",
          "ssh-rsa <klucz RSA> nazarewk@ldap.iem.pw.edu.pl",
          "ssh-rsa <klucz RSA> ato@volt.iem.pw.edu.pl"
        ]
      }
    ]
  },
  "storage": {},
  "systemd": {}
}

```

B.2 repozytorium ipxe-boot

ipxe-boot (<https://github.com/nazarewk/ipxe-boot>) jest repozytorium kodu umożliwiające uruchomienie środowiska bezdyskowego poza uczelnią.

Repozytorium nie zostało wykorzystane w finalnej konfiguracji, więc nie będę opisywał jego zawartości.