Architektura klastra: płaszczyzna sterowania i węzły

Poziom: podstawy
Dotyczy: Kubernetes 1.37, containerd jako środowisko uruchomieniowe **Czas: ~25 min**

Po przeczytaniu tej strony wiemy, z jakich procesów składa się klaster, co robi każdy z nich, co się dzieje, gdy któryś z nich padnie, i jaką drogę przechodzi żądanie od `kubectl apply` do działającego kontenera.

Skróty użyte na tej stronie (11)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CPU	Central Processing Unit: procesor
CRI	Container Runtime Interface: standard, którym Kubernetes rozmawia z silnikiem kontenerów
etcd	etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft
FAQ	Frequently Asked Questions: zbiór najczęściej zadawanych pytań z odpowiedziami
HTTPS	HTTP zabezpieczony szyfrowaniem TLS
JSON	JavaScript Object Notation: tekstowy format danych z klamrami i cudzysłowami
OCI	Open Container Initiative: standard formatu obrazów kontenerów i rejestrów, z którego korzystają też rejestry wykresów Helm (adresy zaczynające się od oci://)
RBAC	Role-Based Access Control: uprawnienia nadawane rolom, a role ludziom
REST	sposób budowania API przez zwykłe żądania HTTP (GET, POST) zamiast osobnego protokołu
YAML	YAML Ain't Markup Language: format plików konfiguracyjnych oparty na wcięciach

Fiszka

TYLKO ODCZYT

Wylistowanie węzłów klastra z rolami i wersjami: kubectl get nodes -o wide

```
kubectl get nodes -o wide
```

Zwracamy listę węzłów należących do klastra wraz z ich rolą, statusem gotowości, wersją `kubelet` i używanym środowiskiem uruchomieniowym kontenerów.

Przykład:

```
NAME    STATUS   ROLES           AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-
IMAGE             KERNEL-VERSION       CONTAINER-RUNTIME
k3s01   Ready    control-plane   62d   v1.37.1   10.20.10.44   <none>        Ubuntu
22.04.4 LTS   5.15.0-131-generic   containerd://1.7.22
k3s02   Ready    <none>          62d   v1.37.1   10.20.10.67   <none>        Ubuntu
22.04.4 LTS   5.15.0-131-generic   containerd://1.7.22
k3s03   Ready    <none>          62d   v1.37.1   10.20.10.68   <none>        Ubuntu
22.04.4 LTS   5.15.0-131-generic   containerd://1.7.22
```

> Laboratorium Zielonej to k3s: węzły `k3s01` (10.20.10.44), `k3s02` (10.20.10.67) i `k3s03` (10.20.10.68). Sposób budowy klastra przez `kubeadm` opisujemy jako ogólny, dlatego listy podów niżej pokazują układ typowy dla `kubeadm`, z nazwami węzłów z naszego laboratorium.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Wylistowanie podów płaszczyzny sterowania: kubectl get pods -n kube-system

```
kubectl get pods -n kube-system -o wide
```

Na klastrze postawionym przez `kubeadm` większość składników płaszczyzny sterowania działa jako zwykłe pody, uruchamiane bezpośrednio przez `kubelet` z manifestów statycznych, a nie przez planistę — dzięki temu widzimy je tą samą komendą, którą sprawdzalibyśmy dowolny inny pod.

Przykład:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
etcd-k3s01	1/1	Running	0	62d	10.20.10.44	k3s01
kube-apiserver-k3s01	1/1	Running	0	62d	10.20.10.44	k3s01
kube-controller-manager-k3s01	1/1	Running	2	62d	10.20.10.44	k3s01
kube-scheduler-k3s01	1/1	Running	1	62d	10.20.10.44	k3s01
kube-proxy-4f2xz	1/1	Running	0	62d	10.20.10.44	k3s01
kube-proxy-9jd7q	1/1	Running	0	62d	10.20.10.67	k3s02
kube-proxy-p2kmn	1/1	Running	0	62d	10.20.10.68	k3s03
coredns-7db6d8ff4d-2xqvz	1/1	Running	0	62d	10.244.0.5	k3s01
coredns-7db6d8ff4d-8mnzs	1/1	Running	0	62d	10.244.0.6	k3s01

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

O 14:40 Tomek wykonuje `kubectl apply` na klastrze testowym i przez dwie minuty nic się nie dzieje, więc zaczyna podejrzewać zawieszony proces. Sprawdza `kubectl get pods -n kube-system -o wide` i widzi, że `kube-scheduler` ma status `CrashLoopBackOff` po nieudanej aktualizacji obrazu na węźle sterującym. Nowy pod wisi w fazie `Pending`, bo nikt nie decyduje, na który węzeł roboczy go przydzielić, a dopiero po przywróceniu poprzedniej wersji obrazu planisty pod dostaje przydział i rusza.

Klaster, węzeł sterujący i węzeł roboczy

Pojęcie: klaster. Zbiór maszyn (fizycznych albo wirtualnych), które razem tworzą jeden Kubernetesowy system — z zewnątrz widziany i zarządzany jako jedna całość przez jedno API, mimo że w środku pracuje wiele niezależnych procesów rozłożonych na wielu hostach.

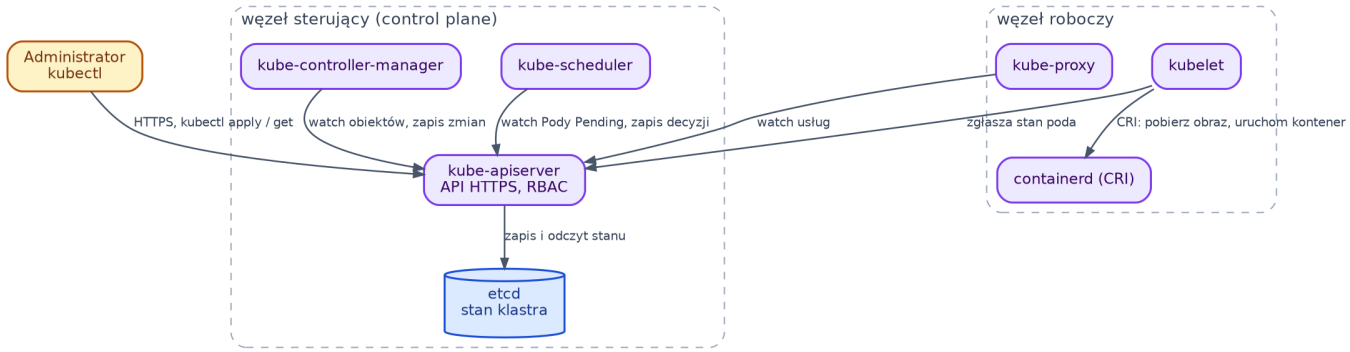
> **Pojęcie: węzeł (node).** Pojedyncza maszyna należąca do klastra. Kubernetes rozróżnia dwie role węzła, które mogą, ale nie muszą, być rozdzielone na osobne maszyny.

Węzeł sterujący (control-plane node) uruchamia procesy odpowiedzialne za zarządzanie całym klastrem — podejmowanie decyzji, przechowywanie stanu, reagowanie na zmiany. Na małych klastrach testowych (patrz sekcja o klastrach lokalnych na stronie kubectl: narzędzie codzienne) bywa jednocześnie jedynym węzłem, na którym da się też uruchamiać zwykłe aplikacje; na produkcji zwykle jest odseparowany i nie przyjmuje obciążenia aplikacyjnego.

Węzeł roboczy (worker node) uruchamia rzeczywiste kontenery naszych aplikacji. Nie podejmuje samodzielnie decyzji o tym, co ma uruchomić — czeka, aż płaszczyzna sterowania mu to przydzieli, i wykonuje.

Składniki płaszczyzny sterowania

> **Pojęcie: płaszczyzna sterowania (control plane).** Zestaw procesów pełniących rolę mózgu klastra — przyjmują żądania, przechowują pożądany i aktualny stan każdego obiektu, podejmują decyzje o przydziale zasobów i uruchamiają pętlę uzgadniania opisaną na stronie Po co Kubernetes, skoro mamy Dockera. Żaden z tych procesów sam nie uruchamia kontenerów naszej aplikacji — tym zajmują się wyłącznie węzły robocze.



Węzeł sterujący trzyma apiserver, etcd, scheduler i controller-manager, a węzeł roboczy kubelet, kube-proxy i containerd, przy czym wszystkie procesy rozmawiają wyłącznie przez apiserver (kliknięcie otwiera pełny rozmiar)

kube-apiserver

Jedyny punkt wejścia do klastra. Wszystko, co dzieje się w Kubernetesie — `kubectl apply`, odczyt stanu przez `kubectl get`, komunikacja kontrolerów między sobą, zgłoszenia `kubelet` o stanie podów — przechodzi przez ten jeden proces, wystawiony jako REST API po HTTPS. `kube-apiserver` uwierzytelnia żądanie (certyfikat klienta, token, moduł zewnętrzny), sprawdza autoryzację (RBAC), przepuszcza żądanie przez kontrolery przyjęcia (admission controllers, w tym walidację standardów bezpieczeństwa poda, patrz niżej) i dopiero wtedy zapisuje albo odczytuje dane z `etcd`. Jest bezstanowy — nie trzyma niczego wyłącznie we własnej pamięci — dlatego można go swobodnie skalować poziomo za load balancerem.

Co się dzieje, gdy padnie: żadne nowe żądanie nie przejdzie — `kubectl` przestaje odpowiadać, kontrolery nie widzą zmian i nie mogą reagować na awarie, `kubelet` na każdym węźle traci możliwość zgłaszania stanu i pobierania nowych przydziałów. Pody, które już działały w momencie awarii, nie znikają — `kubelet` na każdym węźle kontynuuje pilnowanie ich samodzielnie, na podstawie ostatniego znanego przydziału — ale klaster przestaje reagować na cokolwiek nowego, dopóki `kube-apiserver` nie wróci.

etcd

> **Pojęcie: etcd.** Rozproszona baza danych typu klucz-wartość, w której `kube-apiserver` przechowuje kompletny stan klastra — każdy obiekt, jego `spec` i `status`, w jednym miejscu. Nic w Kubernetesie nie trzyma stanu poza `etcd` — nawet planista i kontrolery przy każdym restarcie odtwarzają swoją wiedzę o świecie, czytając ją z powrotem z `etcd` przez `kube-apiserver`.

Co się dzieje, gdy padnie: `kube-apiserver` przestaje móc odczytywać i zapisywać stan, więc w praktyce klaster przestaje przyjmować jakiekolwiek zmiany, nawet jeśli sam `kube-apiserver` formalnie odpowiada na żądania. Utrata wszystkich kopii `etcd` bez kopii zapasowej oznacza utratę całej wiedzy o tym, co powinno działać w klastrze — kontenery na węzłach fizycznie nadal pracują, ale nikt już nie wie, że powinny.

kube-scheduler

> **Pojęcie: planista (scheduler).** Proces, który obserwuje nowo utworzone pody bez przypisanego węzła (pole `nodeName` puste) i decyduje, na którym węźle każdy z nich ma zostać uruchomiony. Decyzja bierze pod uwagę dostępne zasoby węzła (ile CPU i pamięci już jest zarezerwowane przez inne pody), reguły przypisania (affinity/anti-affinity), taints i tolerations oraz inne ograniczenia zadeklarowane w manifeście. Planista nie uruchamia niczego sam — jedynie zapisuje w `etcd` decyzję „ten pod należy do tego węzła", a wykonanie przejmuje `kubelet` na wskazanym węźle.

Co się dzieje, gdy padnie: pody już przypisane do węzłów działają dalej bez zmian. Nowe pody utworzone w tym czasie zostają w fazie `Pending` — istnieją jako obiekty w `etcd`, ale nikt nie decyduje, gdzie mają trafić, dopóki planista nie wróci i nie nadrobi zaległości.

kube-controller-manager

Pojedynczy proces, wewnątrz którego działa w rzeczywistości kilkanaście niezależnych kontrolerów — kontroler węzłów (zauważa, gdy węzeł przestaje odpowiadać), kontroler ReplicaSet (pilnuje liczby replik, patrz strona Deployment, ReplicaSet i aktualizacje bez przerwy), kontroler zadań (`Job` / `CronJob`) i wiele innych. Każdy z nich realizuje osobną pętlę uzgadniania dla swojego rodzaju obiektu, ale wszystkie działają w ramach tego samego procesu.

Co się dzieje, gdy padnie: żadna pętla uzgadniania nie działa. Istniejące pody nadal pracują, ale awaria węzła nie zostanie wykryta i skompensowana nowym podem gdzie indziej, ręczne skalowanie przez zmianę `replicas` w `etcd` (teoretycznie, przez bezpośredni zapis) nie zostanie zrealizowane, a zakończone zadania `Job` nie zostaną posprzątane.

cloud-controller-manager

Składnik opcjonalny, obecny wyłącznie na klastrach uruchomionych w chmurze publicznej albo zintegrowanych z konkretnym dostawcą infrastruktury. Odseparowuje logikę specyficzną dla dostawcy (tworzenie load balancerów przy typie usługi `LoadBalancer`, przydzielanie woluminów blokowych, oznaczanie węzła jako niedostępnego, gdy odpowiadająca mu maszyna wirtualna zniknęła z chmury) od reszty `kube-controller-manager`, która ma pozostać identyczna niezależnie od tego, gdzie klaster działa. Klaster postawiony samodzielnie na gołym metalu (`kubeadm` na własnych serwerach) zwykle w ogóle go nie uruchamia.

Co się dzieje, gdy padnie: funkcje integracji z chmurą (nowe load balancery, nowe woluminy) przestają się realizować, reszta klastra działa bez zmian.

Składniki węzła

kubelet

> **Pojęcie: kubelet.** Agent działający na każdym węźle (sterującym i roboczym), który utrzymuje stałe połączenie z `kube-apiserver`, obserwuje, jakie pody zostały przypisane do jego węzła, i dba o to, żeby faktycznie działały — pobiera obrazy, uruchamia kontenery przez

> środowisko uruchomieniowe, wykonuje sondy (patrz strona Pod: najmniejsza jednostka), restartuje kontenery zgodnie z `restartPolicy` i zgłasza z powrotem do `kube-apiserver` bieżący stan każdego poda. `kubelet` to jedyny składnik płaszczyzny sterowania, który ma bezpośredni kontakt z rzeczywistością na danym węźle — wszystko inne działa na poziomie deklaracji w `etcd`.

kube-proxy

Proces sieciowy działający na każdym węźle, odpowiedzialny za realizację obiektów `Service` — utrzymuje reguły przekierowania ruchu (przez `iptables` albo `IPVS`, w zależności od trybu) tak, żeby ruch skierowany na stały adres usługi trafiał do jednego z aktualnie działających podów za tą usługą. Szczegóły mechanizmu sieciowego i typów usług rozwija część poświęcona sieci klastra — tutaj wystarczy wiedzieć, że to `kube-proxy` sprawia, że adres usługi działa niezależnie od tego, które konkretnie pody aktualnie za nią stoją.

Środowisko uruchomieniowe kontenerów i los Dockershim

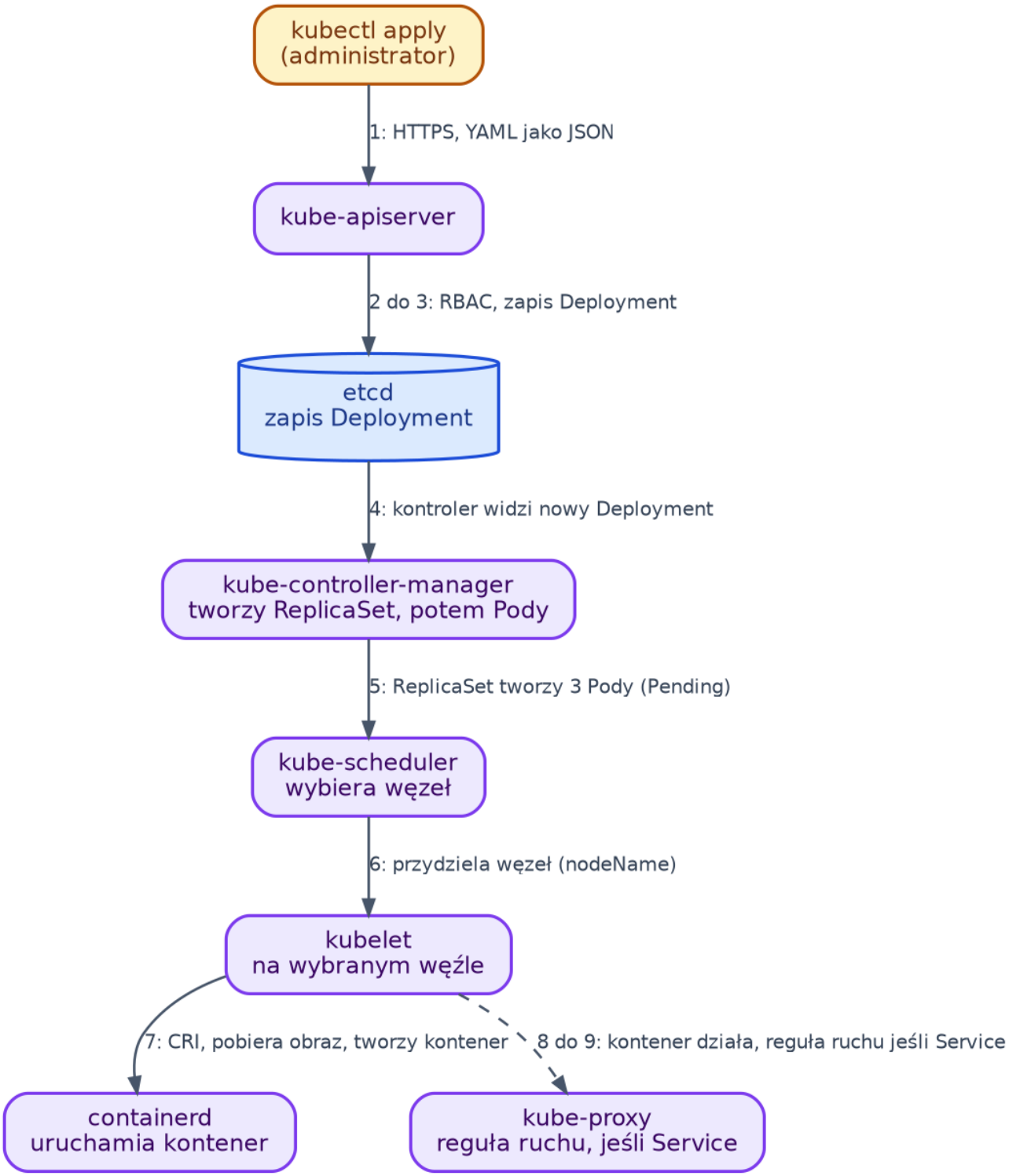
> **Pojęcie: CRI (Container Runtime Interface).** Standardowy interfejs, przez który `kubelet` komunikuje się z dowolnym środowiskiem uruchomieniowym kontenerów, nie wiążąc się na stałe z konkretną implementacją. Dzięki temu `kubelet` obsługuje `containerd`, `CRI-O` czy dowolne inne środowisko zgodne z CRI, wywołując dokładnie te same operacje — utwórz kontener, uruchom, zatrzymaj, pobierz obraz.

Docker Engine nigdy nie mówił natywnie w CRI — powstał wcześniej niż ten standard. Żeby Kubernetes mógł używać Dockera jako środowiska uruchomieniowego, `kubelet` przez lata ładował dodatkową warstwę tłumaczącą, `dockershim`, która przekładała wywołania CRI na wywołania API Dockera. Ta warstwa została usunięta z `kubelet` w wydaniu **Kubernetes 1.24** (kwiecień 2022) — od tej wersji Docker Engine przestał być obsługiwanym środowiskiem uruchomieniowym węzła.

To nie oznacza, że obrazy budowane Dockerem przestały działać w Kubernetesie — obrazy w formacie OCI, które produkuje `docker build`, są identyczne niezależnie od tego, jaki silnik je uruchamia, i `containerd` czy `CRI-O` uruchamiają je bez żadnej różnicy. Zniknął tylko sam demon Dockera jako proces zarządzający kontenerami na węźle roboczym — w jego miejsce większość dystrybucji Kubernetesa (w tym `kubeadm`) domyślnie instaluje `containerd`, czyli w istocie tę samą bibliotekę wykonawczą, na której Docker Engine sam był zbudowany.

Droga żądania: od kubectl apply do działającego kontenera

To jest najważniejszy fragment tej strony — spina wszystkie powyższe składniki w jeden przebieg. Zakładamy, że wykonujemy `kubectl apply -f wdrozenie.yaml`, gdzie manifest opisuje Deployment z trzema replikami.



Żądanie kubectl apply przechodzi kolejno przez apiserver, etcd, controller-manager, scheduler, kubelet i containerd, aż do działającego kontenera (kliknięcie otwiera pełny rozmiar)

1. **`kubectl`** wczytuje plik YAML, serializuje go do JSON i wysyła żądanie HTTPS do `kube-apiserver`, dołączając poświadczenia z aktualnego kontekstu w `~/.kube/config` (temat rozwija strona kubectl: narzędzie codzienne).

2. **`kube-apiserver`** uwierzytelnia żądanie i sprawdza w RBAC, czy to konto ma prawo tworzyć obiekty typu Deployment w tej przestrzeni nazw. Następnie przepuszcza obiekt przez kontrolery przyjęcia — walidujące (np. zgodność ze standardami bezpieczeństwa poda) i mutujące (mogą dopisać domyślne wartości) — i waliduje zgodność ze schematem danego rodzaju zasobu.
3. Obiekt Deployment trafia do **`etcd`** jako nowy wpis — to jest jedyny moment, w którym cokolwiek zostaje trwale zapisane.
4. Kontroler Deploymentu wewnątrz **`kube-controller-manager`** obserwuje (mechanizmem `watch` na `kube-apiserver`) nowo utworzone obiekty Deployment. Widząc nowy, tworzy odpowiadający mu obiekt ReplicaSet z `replicas: 3`.
5. Kontroler ReplicaSet, również wewnątrz `kube-controller-manager`, widzi nowy ReplicaSet bez żadnych podów i tworzy trzy obiekty Pod — na razie bez przypisanego węzła (`nodeName` puste, faza `Pending`).
6. **`kube-scheduler`** obserwuje pody bez przypisanego węzła. Dla każdego z trzech ocenia dostępne węzły robocze pod kątem wolnych zasobów i reguł przypisania, wybiera najlepszy i zapisuje decyzję z powrotem przez `kube-apiserver` do `etcd` — uzupełniając pole `nodeName`.
7. **`kubelet`** na wskazanym węźle, obserwujący pody przypisane do siebie, widzi nowy pod. Każe środowisku uruchomieniowemu (**`containerd`** przez CRI) pobrać wskazany obraz z rejestru, jeśli go jeszcze nie ma lokalnie, i utworzyć kontener — z własnymi przestrzeniami nazw jądra i `cgroups`, dokładnie tak jak przy `docker run`, tylko zainicjowane przez `kubelet`, a nie ręcznie.
8. Kontener startuje. `kubelet` zaczyna wykonywać zadeklarowane sondy (startową, gotowości, żywotności) i cyklicznie zgłasza aktualny stan poda z powrotem do `kube-apiserver`, który zapisuje go w `etcd` jako pole `status`.
9. Jeśli pod jest celem jakiegoś obiektu `Service`, **`kube-proxy`** na każdym węźle dostaje przez `watch` informację o nowym gotowym podzie i dokłada regułę przekierowania ruchu do niego.

Ten sam przebieg, w odwrotną stronę, tłumaczy dlaczego `kubectl get pods` zaraz po `apply` pokazuje pody w fazie `Pending`, a dopiero po chwili `Running` — obserwujemy krok po kroku tę samą sekwencję zdarzeń, którą właśnie opisaliśmy, tylko oglądaną z zewnątrz.

Wysoka dostępność płaszczyzny sterowania

Pojedynczy węzeł sterujący jest pojedynczym punktem awarii dla wszystkiego, co wymaga podjęcia nowej decyzji — dlatego produkcyjne klastry stawia się z co najmniej trzema węzłami sterującymi.

`etcd` działa wtedy jako klaster rozproszony, w którym zapis wymaga zgody większości członków (kworum) — stąd nieparzysta liczba węzłów (3 albo 5), bo przy parzystej liczbie łatwiej o sytuację,

w której żadna ze stron podziału sieci nie ma większości. `kube-apiserver` uruchamiamy na każdym węźle sterującym niezależnie i stawiamy przed nimi load balancer — każda replika jest bezstanowa i w pełni równoprawna, więc żaden dodatkowy mechanizm wyboru lidera nie jest jej potrzebny. `kube-scheduler` i `kube-controller-manager` natomiast muszą działać jako dokładnie jedna aktywna instancja naraz (uruchomienie dwóch planistów decydujących niezależnie o tym samym podzie prowadziłoby do konfliktów) — dlatego każda replika rywalizuje o dzierżawę (lease) w `etcd` metodą leader election, i tylko ta, która ją aktualnie trzyma, faktycznie działa; pozostałe stoją w gotowości i przejmują rolę natychmiast, gdy lider przestanie odnawiać dzierżawę.

Częste błędy

Zakładanie, że awaria `kube-apiserver` natychmiast zatrzymuje działające aplikacje. Pody, które już działały, kontynuuje `kubelet` na swoim węźle niezależnie od dostępności płaszczyzny sterowania — przestaje działać wyłącznie zdolność do wprowadzania nowych zmian i reagowania na nowe awarie.

Mylenie kopii zapasowej węzła sterującego z kopią zapasową `etcd`. Stan całego klastra leży w `etcd`, nie w systemie plików węzła sterującego — kopia zapasowa `etcd` (opisana osobno w części o eksploatacji) jest tym, co faktycznie pozwala odtworzyć klaster po katastrofie.

Przekonanie, że usunięcie Dockera z węzłów oznacza koniec obsługi obrazów budowanych Dockerem. Znika demon Dockera jako środowisko uruchomieniowe węzła, nie format obrazu — `docker build` i `docker push` na stacji roboczej administratora działają bez zmian, zmienia się wyłącznie to, czym te obrazy są uruchamiane na węzłach klastra.

□□ Ćwiczenia

> Zadania sprawdzające.

1. Opisujemy, co dokładnie przestaje działać w klastrze, gdy pada jedyny węzeł sterujący, a co nadal działa bez zmian.
2. Wymieniamy pięć składników płaszczyzny sterowania i dla każdego podajemy jedno zdanie o tym, co się stanie, jeśli akurat on ulegnie awarii.
3. Prześledzamy krok po kroku, co się dzieje od momentu wykonania `kubectl apply -f manifest.yaml` do momentu, gdy `kubectl get pods` pokazuje status `Running` — z wymienieniem, który składnik odpowiada za który krok.
4. Wyjaśniamy, dlaczego usunięcie `dockershim` w Kubernetes 1.24 nie unieważniło obrazów zbudowanych wcześniej poleceniem `docker build`.

Powiązane: Po co Kubernetes, skoro mamy Dockera · Obiekty, manifesty i przestrzenie nazw · Pod: najmniejsza jednostka · kubectl: narzędzie codzienne

Źródła: dokumentacja Kubernetes, rozdział „Kubernetes Components" (kubernetes.io/docs/concepts/overview/components/) · „Container Runtimes" (kubernetes.io/docs/setup/production-environment/container-runtimes/) · „Dockershim Removal FAQ" (kubernetes.io/blog/2022/02/17/dockershim-faq)

Revision #34
Created 2026-09-25 10:32:51 UTC by Tomasz Konieczny
Updated 2026-10-26 14:16:52 UTC by Tomasz Konieczny