k3s z trzema węzłami serwerowymi: etcd, adres wirtualny API i Longhorn

Poziom: zaawansowane

Dotyczy: k3s v1.36.5+k3s1 (Kubernetes 1.36, wbudowany etcd), Longhorn 1.13.0, Rocky Linux 9, Hyper-V na hv01 i hv02, HAProxy i keepalived ze strony o równoważeniu

Czas: ~150 min

Rozbudowujemy laboratoryjny k3s z jednego węzła `k3s01` do trzech serwerów z wbudowanym etcd i z adresem wirtualnym API. Pokazujemy, dlaczego trzy węzły na dwóch hostach Hyper-V mają jedno słabe miejsce, i jak je ograniczyć. Na koniec sprawdzamy dane na Longhornie i tym samym kolorowym nginksie co wcześniej, wyłączając węzły po kolei.

ZAŁOŻENIA SCENARIUSZA

trzy węzły stoją na dwóch hostach Hyper-V: `k3s01` i `k3s03` na `hv01`, a `k3s02` na `hv02`, w VLAN 10 i w jednej serwerowni. Awaria `hv01` zabiera dwa głosy etcd z trzech do czasu restartu maszyn na `hv02`.

Skróty użyte na tej stronie (19)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
DNS	Domain Name System: książka telefoniczna internetu, zamienia nazwy na adresy IP
etcd	etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft
GB	Gigabyte: miliard bajtów, jednostka pojemności pamięci i plików, tu najczęściej rozmiar modelu albo pamięci karty graficznej
GiB	gibibajt: 1024 MiB, czyli około 1,07 GB; tej jednostki używa dokumentacja MailCow przy wymaganiach pamięci

Skrót	Co znaczy
HAProxy	High Availability Proxy: program, który rozdziela połączenia między kilka serwerów i omija te, które nie odpowiadają
HTTP	Hypertext Transfer Protocol: protokół stron WWW
PVC	PersistentVolumeClaim: zamówienie trwałego dysku przez aplikację w Kubernetesie
Raft	protokół uzgadniania stanu w klastrze: zmiana jest przyjęta, gdy zgodzi się większość węzłów; na nim stoją menedżerowie Swarm i etcd
S2D	Storage Spaces Direct: programowa macierz Windows Server z dysków wielu serwerów
SAN	w certyfikatach: Subject Alternative Name, dodatkowe nazwy hosta w certyfikacie; w pamięciach: Storage Area Network, sieć dyskowa
SELinux	Security-Enhanced Linux: dodatkowa warstwa uprawnień w jądrze RHEL
SNI	Server Name Indication: pole w pierwszym komunikacie TLS, w którym klient podaje nazwę serwera, z którym chce rozmawiać, co pozwala HAProxy wybrać backend bez rozszyfrowywania ruchu
SQLite	mała baza danych zapisana w jednym pliku; Home Assistant trzyma w niej historię w pliku home-assistant_v2.db
SSD	Solid State Drive: dysk półprzewodnikowy
TLS	Transport Layer Security: szyfrowanie połączeń, np. w HTTPS
VLAN	Virtual LAN: wydzielona logicznie sieć na tym samym sprzęcie
VRRP	Virtual Router Redundancy Protocol: protokół, w którym kilka hostów uzgadnia, który z nich trzyma wspólny adres IP (opisany w RFC 5798)

Fiszka

Komendy wykonujemy z CLI bash z uprawnieniami root (sudo) na węźle k3s, chyba że przy komendzie napisano inaczej. Polecenia `kubectl` uruchamiamy na dowolnym serwerze albo na stacji administratora ze skopiowanym plikiem kubeconfig (jak na stronie k3s w laboratorium: pierwszy deploy, nginx w kolorach).

Adresy: `k3s01` to `10.20.10.44`, `k3s02` to `10.20.10.67`, `k3s03` to `10.20.10.68`, adres wirtualny API to `10.20.10.69`, a adres wirtualny równoważenia ruchu do aplikacji to `10.20.10.66`. Bloki „Przykład” pokazują kształt wyniku i nie pochodzą z prób na trzech maszynach Zielonej, a to, co sprawdziliśmy sami, opisuje sekcja „Co sprawdziliśmy w piaskownicy”.

TYLKO ODCZYT

Odczyt wersji k3s na węźle: k3s --version

```
k3s --version
```

Wypisuje wersję k3s i Kubernetesa. Na wszystkich serwerach klastra ma być dokładnie ta sama wersja, więc zaczynamy od odczytu na `k3s01` i taką samą instalujemy na pozostałych.

Przykład:

```
k3s version v1.36.5+k3s1 (a1b2c3d4)
```

k3s = lekka dystrybucja Kubernetesa (5 liter mniej niż k8s)

ZMIENIA SYSTEM

Przejście pierwszego serwera z SQLite na etcd: cluster-init

```
sudo systemctl restart k3s
```

> (gdzie wcześniej do `/etc/rancher/k3s/config.yaml` na `k3s01` dopisaliśmy linię `cluster-init: true`; k3s przy starcie zamienia bazę SQLite na wbudowany etcd, w tym kroku nie ma powrotu do SQLite)

SQLite = mała baza danych zapisana w jednym pliku; Home Assistant trzyma w niej historię w pliku home-assistant_v2.db · **etcd** = etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft · **systemctl** = system control: sterowanie usługami systemd

TYLKO ODCZYT

Odczyt tokenu klastra: cat server/token

```
sudo cat /var/lib/rancher/k3s/server/token
```

Wypisuje token potrzebny kolejnym serwerom do dołączenia. Token szyfruje też poufne dane w bazie, więc trzymamy go w menedżerze haseł i w kopii zapasowej obok migawek etcd.

etcd = etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft · **cat** = concatenate: sklej i wypisz pliki

TYLKO ODCZYT

Dołączenie kolejnego serwera: instalator z config.yaml

```
curl -sfL https://get.k3s.io | INSTALL_K3S_VERSION='WersjaK3s' sh -s - server
```

> (gdzie `WersjaK3s` to wersja z `k3s --version` na `k3s01`, na przykład `v1.36.5+k3s1`; parametry `server` i `token` instalator czyta z `/etc/rancher/k3s/config.yaml`, który tworzymy przed uruchomieniem)

curl = client URL: pobieranie i wysyłanie przez HTTP z terminala

TYLKO ODCZYT

Lista węzłów klastra: kubectl get nodes

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

Pokazuje węzły, ich stan, role i wersję. Trzy wiersze `Ready` z rolami `control-plane` i `etcd` oznaczają klaster trzyserwerowy.

Przykład:

```
NAME    STATUS   ROLES                       AGE   VERSION        INTERNAL-IP
k3s01   Ready    control-plane,etcd,master   12d   v1.36.5+k3s1   10.20.10.44
k3s02   Ready    control-plane,etcd,master   5m    v1.36.5+k3s1   10.20.10.67
k3s03   Ready    control-plane,etcd,master   3m    v1.36.5+k3s1   10.20.10.68
```

etcd = etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie zdrowia API razem z etcd: kubectl get --raw

```
kubectl get --raw='/readyz?verbose' | grep etcd
```

Wypisuje linię kontroli gotowości etcd z perspektywy API. Zdrowy węzeł pokazuje `[+]etcd ok`.

API = Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą · **etcd** = etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft · **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Zapis migawki etcd: k3s etcd-snapshot save

```
sudo k3s etcd-snapshot save --name przed-aktualizacja
sudo k3s etcd-snapshot ls
```

Pierwsza komenda zapisuje migawkę lokalnie w `/var/lib/rancher/k3s/server/db/snapshots`, druga wypisuje migawki. Migawka bez tokenu z karty „Odczyt tokenu klastra” jest bezużyteczna.

etcd = etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft · **k3s** = lekka dystrybucja Kubernetesa (5 liter mniej niż k8s)

OSTROŻNIE · NIEODWRACALNE

Odizolowanie węzła przed pracami: kubectl drain

```
kubectl drain NazwaWezla --ignore-daemonsets --delete-emptydir-data
```

Oznacza węzeł jako niedostępny dla nowych podów i przenosi z niego pody, szanując PodDisruptionBudget. Pody DaemonSetów (Traefik, Longhorn) zostają, bo to ich rola.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Powrót węzła do rotacji: kubectl uncordon

```
kubectl uncordon NazwaWezla
```

Węzeł znów przyjmuje nowe pody. Zadania przeniesione wcześniej nie wracają same, bo scheduler nie przesuwa działających podów bez powodu.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Aktualizacja węzła do wskazanej wersji: INSTALL_K3S_VERSION

```
curl -sfL https://get.k3s.io | INSTALL_K3S_VERSION='NowaWersjaK3s' sh -s - server
```

Instalator podmienia binarkę i restartuje usługę k3s z istniejącego `config.yaml`. Serwery aktualizujemy po jednym, nigdy dwóch naraz.

curl = client URL: pobieranie i wysyłanie przez HTTP z terminala

TYLKO ODCZYT

Stan woluminów Longhorn: kubectl get volumes.longhorn.io

```
kubectl -n longhorn-system get volumes.longhorn.io
```

Pokazuje każdy wolumen, jego stan (`STATE`) i kondycję (`ROBUSTNESS`). Przed każdym kolejnym krokiem prac na węzłach czekamy na `attached` i `healthy` przy wszystkich wolumenach, które mamy podłączone.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

Z ŻYCIA WZIĘTE

Wtorek, 14:05. Tomek pokazuje Oli rysunek: `k3s01` i `k3s03` na `hv01`, `k3s02` na `hv02`, i pyta, czy teraz awaria węzła przestała być problemem. Ola kładzie palec na `hv01` i liczy głosy: dwa węzły z trzech wypadają razem, zostaje jeden głos z trzech, czyli mniej niż większość. Pan Zbigniew, który właśnie wszedł po kawę, pyta, czy to znaczy, że będzie gorzej niż teraz. Ola odpowiada, że nie gorzej, bo dziś pada cały `k3s01` i też nic nie działa do restartu maszyny, ale że trzy węzły chronią przed inną awarią niż dwa hosty.

Jak to działa

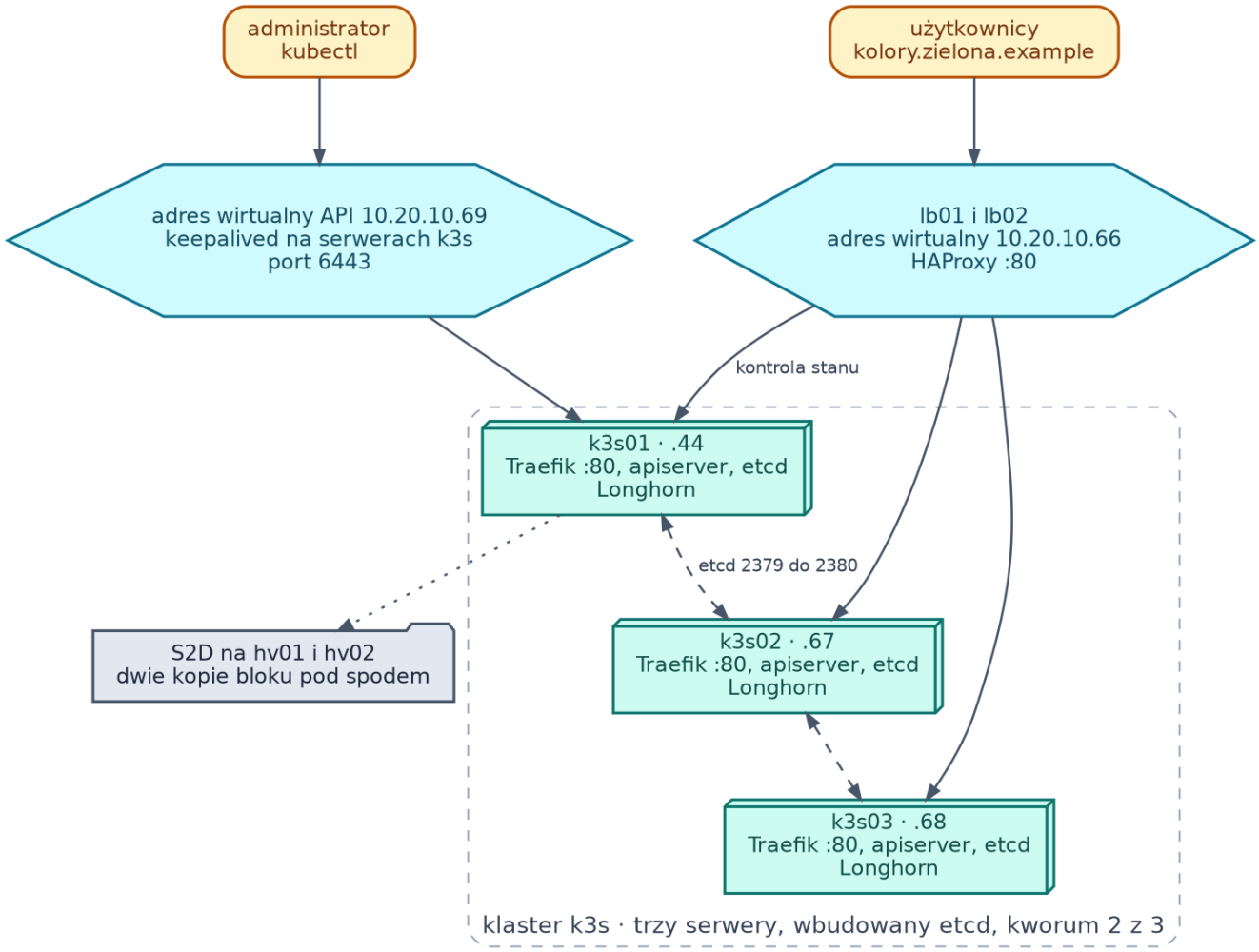
Rozwiń: jak to działa pod spodem

Co daje trzeci serwer

Pojedynczy `k3s01` trzyma stan klastra w lokalnej bazie SQLite. Kiedy maszyna się zatrzymuje, zatrzymuje się razem z nią API, Traefik i wszystkie pody. W układzie z trzema serwerami stan klastra leży we wbudowanym etcd, rozproszonej bazie, która zapisuje każdą zmianę na większości węzłów, zanim uzna ją za zatwierdzoną. Awarię jednego serwera pozostałe dwa przeżywają bez przerwy w pracy API, a Traefik na każdym z węzłów dalej przyjmuje ruch.

Pojęcie: etcd. Rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały swój stan: listę podów, wdrożeń, sekretów i konfiguracji. Do uzgadniania zmian między węzłami używa protokołu Raft, tego samego, na którym stoją menedżerowie Swarm ze strony Docker Swarm: klaster z trzema menedżerami, na którym przenosimy kolorowy nginx. Zmiana jest zatwierdzona, gdy zapisze ją większość członków, więc trzy węzły tolerują utratę jednego, a pięć utratę dwóch.

Pojęcie: serwer i agent w k3s. Serwer to węzeł z płaszczyzną sterowania (API, scheduler, kontrolery) i, w naszym układzie, z członkiem etcd. Agent to węzeł tylko do uruchamiania podów. Wszystkie trzy nasze węzły są serwerami, a jednocześnie uruchamiają zwykłe pody, bo przy trzech maszynach w laboratorium nie oddzielamy ról.



Rysunek 1. Układ k3s: ruch do aplikacji przez adres wirtualny równoważenia, ruch administracyjny przez adres wirtualny API, trzy serwery z etcd i Longhornem (kliknięcie otwiera pełny rozmiar)

Rysunek pokazuje dwa osobne wejścia. Użytkownicy aplikacji idą przez `lb01` i `lb02` pod adres `10.20.10.66` i HAProxy rozdziela ruch na Traefika na każdym z trzech węzłów. Administrator z `kubectl` idzie pod adres `10.20.10.69` na porcie 6443, który trzyma jeden z trzech serwerów.

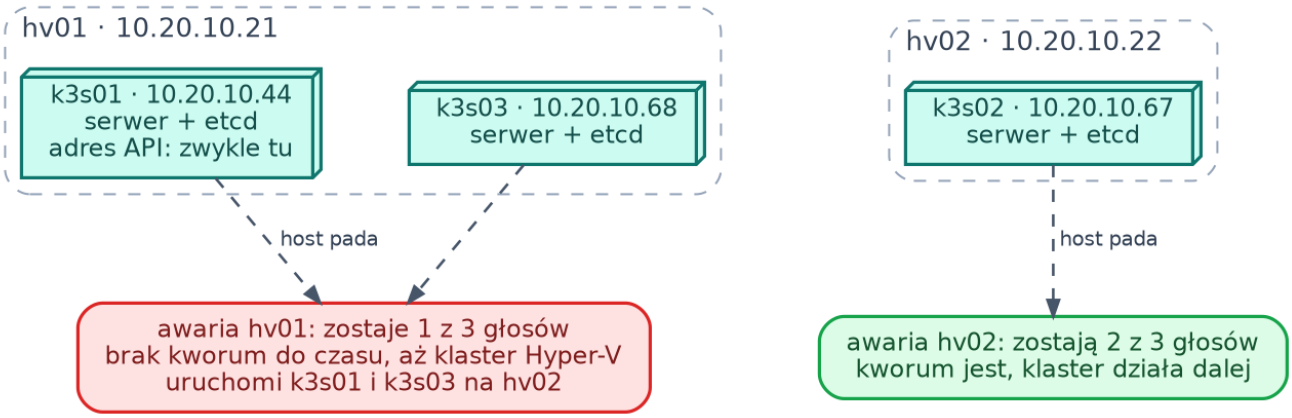
Wersja k3s, którą przyjmujemy

Na 18 października 2026 najnowsze wydania k3s to `v1.37.1+k3s1` i, w gałęzi 1.36, `v1.36.5+k3s1`, oba z 30 września 2026 [6]. Strona o pierwszym wdrożeniu k3s pokazuje w przykładzie `v1.36.4+k3s1` z 27 sierpnia 2026, czyli wcześniejszą poprawkę tej samej gałęzi. Na tej stronie przyjmujemy `v1.36.5+k3s1` i tak ją zapisujemy w komendach, a przejście na gałąź 1.37 opisuje strona k3s w wysokiej dostępności: migawki etcd, wymiana węzła, odtworzenie i aktualizacja.

W laboratorium obowiązuje jedna reguła ważniejsza niż numer: wszystkie serwery mają tę samą wersję. Jeśli `k3s01` działa już na `v1.36.4+k3s1`, to właśnie tę wersję podstawiamy za `WersjaK3s` na `k3s02` i `k3s03`, bo nowsze pliki binarne dołączające do starszego klastra nie są objęte gwarancją zgodności. Numery poprawek sprawdzamy w dniu instalacji na stronie wydań, a w produkcji ustalamy wersję raz i aktualizujemy według sekcji „Aktualizacja węzłów po kolei”.

Trzy węzły na dwóch hostach

Kworum etcd to większość głosów, czyli co najmniej 2 z 3. Mamy jednak dwa hosty Hyper-V, więc któryś z nich musi trzymać dwa węzły. Awaria tego hosta zabiera dwa głosy naraz, a klaster etcd bez kworum zatrzymuje zapisy.



Rysunek 2. Trzy węzły k3s na dwóch hostach: awaria hv02 zostawia kworum, awaria hv01 je zabiera do czasu restartu maszyn na hv02 (kliknięcie otwiera pełny rozmiar)

Na rysunku `k3s01` i `k3s03` stoją na `hv01`, a `k3s02` na `hv02`. Awaria `hv02` zostawia 2 głosy z 3, więc klaster działa dalej. Awaria `hv01` zostawia 1 głos z 3: klaster Hyper-V uruchamia `k3s01` i `k3s03` na `hv02` (maszyny leżą na wspólnym woluminie S2D), kworum wraca, a klaster wstaje sam. Do tego czasu API nie przyjmuje zmian, a istniejące pody na `k3s02` dalej pracują, więc czas restartu mierzymy w teście C.

Wniosek: trzy serwery k3s chronią nas przed awarią jednej maszyny wirtualnej, błędem systemu w jednym węźle i aktualizacją po kolei, ale nie przed awarią hosta. Przed awarią hosta chroni klaster Hyper-V, tak jak dziś. Mamy trzy sposoby, żeby podejść do tego uczciwie.

Wariant	Co daje	Koszt i ograniczenia	Nasza ocena

Trzeci host fizyczny z Hyper-V albo z Linuksem, na którym stoi `k3s03`	po awarii dowolnego hosta kworum zostaje, klaster działa bez przerwy	zakup i utrzymanie sprzętu, trzeci host w klastrze Hyper-V trzeba dołożyć i licencjonować	najlepszy, gdy k3s ma przestać być laboratorium
Węzeł na innym sprzęcie, który już mamy (na przykład na `bkp01` albo `gpu01`, do rozstrzygnięcia przez autora)	to samo co wyżej, bez zakupu	ten sprzęt ma własną rolę, a węzeł etcd zabiera mu pamięć i dysk; węzeł k3s poza klastrem Hyper-V trzeba aktualizować osobno	rozsądny kompromis dla laboratorium, decyzja należy do autora
Układ 2 plus 1 z świadomie przyjętym ryzykiem	zero kosztów, układ z rysunku 2	awaria `hv01` zatrzymuje API do restartu maszyn przez klaster Hyper-V	wystarcza w laboratorium, bo nie jest gorszy od dzisiejszego pojedynczego `k3s01`

Zwracamy uwagę na jeszcze jedną możliwość: k3s potrafi trzymać stan w zewnętrznej bazie (PostgreSQL), ale `pg01` jest pojedynczym serwerem, więc przeniosłoby to problem, a nie rozwiązało go. Dlatego w tym laboratorium przyjmujemy trzeci wariant i opisujemy go uczciwie, a decyzję o drugim lub pierwszym zostawiamy autorowi.

Adres wirtualny API

Plik kubeconfig wskazuje jeden adres serwera. Jeśli to `10.20.10.44`, to po awarii `k3s01` `kubectl` przestaje działać, choć dwa pozostałe serwery są zdrowe. Potrzebujemy adresu, który zawsze należy do żywego serwera. Rozważamy trzy sposoby.

Sposób	Jak działa	Zalety	Wady
keepalived na trzech serwerach	VRRP przenosi adres `10.20.10.69` na serwer ze zdrowym apiserver	nie zależy od Kubernetesa, ten sam mechanizm co na `docker01`, `lb01` i `lb02`	adres siedzi tylko na jednym serwerze, więc ruch nie jest rozkładany
kube-vip jako DaemonSet albo statyczny pod	adres przenosi proces działający w klastrze, wybór lidera przez obiekt Lease w API (według dokumentacji kube-vip)	jedno narzędzie dla adresu API i usług typu LoadBalancer	sam potrzebuje działającego API do wyboru lidera, nowy element do poznania
HAProxy na `lb01` i `lb02` z backendem na 6443	para `lb` rozkłada ruch na trzy serwery	rozkłada ruch, nie dokłada nowego narzędzia	API zależy od pary `lb`, a awaria pary zabiera dostęp do klastra

W Zielonej wybieramy keepalived. Zespół zna go z poprzednich stron, a on sam nie zależy od stanu klastra. Kontrolę stanu robi skrypt, który pyta lokalny apiserver o `/readyz`. Wariant z HAProxy podajemy w sekcji „Wariant: API przez parę lb”.

Ruch do aplikacji

Traefik z k3s jest uruchamiany na każdym z trzech węzłów, a ServiceLB (wbudowany w k3s) otwiera na każdym z nich porty 80 i 443. Dlatego HAProxy może mieć trzy serwery w backendzie `be_k3s_http` i sprawdzać stan każdego osobno. Kontrola z nagłówkiem `Host: kolory.zielona.example` sprawdza więc nie tylko to, że węzeł odpowiada, ale że Traefik na nim zna właściwą regułę Ingress i ma dokąd skierować ruch.

Zwracamy uwagę na ograniczenie: adres klienta, który widzi aplikacja, to adres pary `lb`, dopóki nie skonfigurujemy Traefika z k3s (przez `HelmChartConfig`) tak, żeby ufał nagłówkowi `X-Forwarded-For` z `10.20.10.64/31`. To osobny temat, którego tu nie rozwijamy.

Longhorn i liczba replik na S2D

Pody z danymi potrzebują wolumenu, który przeżyje przeniesienie poda na inny węzeł. Domyślny `local-path` w k3s trzyma dane na dysku jednego węzła, więc pod nie może wrócić na inny. Longhorn rozwiązuje to, trzymając kopie wolumenu (repliki) na kilku węzłach i synchronicznie zapisując każdą zmianę.

Pod spodem mamy jednak S2D z dwukierunkowym lustrem, które już trzyma dwie kopie każdego bloku na dwóch hostach. Liczba replik Longhorna mnoży się przez dwa z S2D.

Liczba replik Longhorna	Zajętość surowa dla 100 GB użytecznych	Co tracimy	Co zyskujemy
1	200 GB	nie ma zapasowej kopii przy awarii węzła k3s, wolumen czeka na powrót węzła	najmniejszy narzut i najkrótszy zapis
2	400 GB	dwukrotnie więcej miejsca i zapisów niż dla jednej repliki	wolumen przeżywa awarię jednego węzła bez czekania, odbudowa idzie na trzeci węzeł
3 (domyślne w Longhornie)	600 GB	trzy kopie na trzech węzłach, brak wolnego węzła na odbudowę	pełna ochrona przy trzech zdrowych węzłach

W Zielonej wybieramy dwie repliki. Ochronę przed awarią dysku daje już S2D, a Longhorn ma chronić przed awarią węzła k3s i przyspieszać przejście poda, bo trzy repliki na trzech węzłach przy jednej awarii nie mają dokąd się odbudować: Longhorn domyślnie nie kładzie dwóch replik jednego wolumenu na tym samym węźle. Jest też druga strona: dwie repliki na dwóch węzłach, z których oba są na `hv01`, nie przeżyją awarii `hv01` bez restartu maszyn, tak samo jak etcd.

Zalecenia Longhorna dla węzłów to 4 vCPU, 4 GiB pamięci, SSD i osobny dysk na dane. W laboratorium kierujemy się nimi tak: 4 vCPU, 8 GiB, dysk systemowy 40 GB i drugi wirtualny dysk 50 GB zamontowany pod `/var/lib/longhorn`.

Przebieg

> W naszym środowisku dane aplikacji w Dockerze trzymamy w `/docker/apps/<nazwa>`, a plik Compose w `/docker/compose/<nazwa>/compose.yaml`. Na tej stronie Dockera nie używamy, ale konwencja ma sens także tutaj: manifesty trzymamy w jednym stałym miejscu, w katalogu `/docker/compose/k3s-kolory` na `k3s01`.

Zanim zaczniemy

Zakładamy, że `k3s01` z kolorowym nginksem działa jak na stronie k3s w laboratorium: pierwszy deploy, nginx w kolorach i że para `lb01` i `lb02` stoi według strony Równoważenie i adres wirtualny: keepalived i HAProxy na parze lb01 i lb02. Potrzebujemy dwóch nowych maszyn z Rocky Linux 9, `k3s02` (`10.20.10.67`) i `k3s03` (`10.20.10.68`), skonfigurowanych jak `k3s01`. Adresy `10.20.10.67` do `10.20.10.69` sprawdziliśmy w całej bazie wiedzy: nie są nigdzie użyte.

Te same trzy adresy wykorzystuje laboratorium Swarm ze strony Docker Swarm: klaster z trzema menedżerami, na którym przenosimy kolorowy nginx, gdzie są to `swarm01` do `swarm03`. Oba laboratoria stawiamy po kolei, nie równocześnie. Maszyny `swarm02` i `swarm03` po zakończeniu laboratorium Swarm albo instalujemy od nowa jako `k3s02` i `k3s03`, albo kasujemy i zakładamy świeże. Adres `10.20.10.69` w laboratorium k3s staje się adresem wirtualnym API (nie osobną maszyną).

Czas w klastrze etcd ma znaczenie, więc na wszystkich trzech węzłach sprawdzamy synchronizację przez `chronyc tracking`, zanim zaczniemy. Przed pracą zapisujemy też stan wyjściowy: `kubectl get nodes -o wide` i `k3s --version` z `k3s01`.

1. Rozmieszczenie maszyn na hv01 i hv02

Na klastrze Hyper-V (komendy z `dc01` albo z hosta, w PowerShellu administratora) ustawiamy preferowanych właścicieli, tak żeby `k3s01` i `k3s03` startowały na `hv01`, a `k3s02` na `hv02`:

```
Set-ClusterOwnerNode -Group "k3s01" -Owners hv01,hv02
Set-ClusterOwnerNode -Group "k3s03" -Owners hv01,hv02
```

```
Set-ClusterOwnerNode -Group "k3s02" -Owners hv02,hv01
```

Kolejność na liście to kolejność preferencji, a druga pozycja pozwala uruchomić maszynę na drugim hoście po awarii pierwszego. Maszyny muszą być rolami klastra, tak jak `docker01`. Jak je dodać do klastra, opisuje strona Hyper-V: maszyny wirtualne, punkty kontrolne i migracja na żywo.

◫ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: Menedżer klastra z trzema rolami `k3s01`, `k3s02` i `k3s03` oraz kolumną „Węzeł właściciela” pokazującą `hv01`, `hv02` i `hv01`.

2. Zapora firewalld na każdym węźle

k3s na Rocky Linux 9 sam instaluje politykę SELinux (`k3s-selinux`), więc SELinux zostaje w trybie `enforcing`. Dokumentacja k3s zaleca wyłączenie firewalld albo otwarcie wskazanych portów, a my wybieramy otwarcie, w strefie przeznaczonej tylko dla węzłów klastra. Wykonujemy na wszystkich trzech węzłach:

```
sudo firewall-cmd --permanent --new-zone=k3s
for ip in 10.20.10.44 10.20.10.67 10.20.10.68; do
  sudo firewall-cmd --permanent --zone=k3s --add-source=$ip
done
sudo firewall-cmd --permanent --zone=k3s --add-port=6443/tcp
sudo firewall-cmd --permanent --zone=k3s --add-port=2379-2380/tcp
sudo firewall-cmd --permanent --zone=k3s --add-port=8472/udp
sudo firewall-cmd --permanent --zone=k3s --add-port=10250/tcp
sudo firewall-cmd --permanent --zone=k3s --add-protocol=vrrp
sudo firewall-cmd --permanent --zone=trusted --add-source=10.42.0.0/16
sudo firewall-cmd --permanent --zone=trusted --add-source=10.43.0.0/16
sudo firewall-cmd --reload
```

Adresy pod i usług (`10.42.0.0/16` i `10.43.0.0/16`) to domyślne zakresy k3s, oznaczone w strefie `trusted`, żeby ruch między podami nie trafiał na zaporę hosta. Protokół `vrrp` jest potrzebny dla keepalived z kroku 5. Dla Longhorna dochodzą porty wymienione w jego dokumentacji, a w laboratorium sprawdzamy, czy strefa `k3s` je pokrywa.

Dla ruchu z zewnątrz otwieramy w strefie domyślnej tylko to, co potrzebne:

```
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source
address="10.20.10.64/31" port port="80" protocol="tcp" accept'
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source
```

```
address="10.20.10.0/24" port port="6443" protocol="tcp" accept'
sudo firewall-cmd --reload
```

Pierwsza reguła wpuszcza HAProxy (`lb01` i `lb02`) na port 80 Traefika, druga pozwala każdej maszynie z VLAN 10 sięgnąć do API przez adres wirtualny. Port 443 zostaje zamknięty: kolorowy nginx działa w laboratorium po HTTP, jak na stronie o pierwszym wdrożeniu. Gdy dodamy TLS, dołożymy analogiczną regułę dla portu 443 i backend w trybie `tcp` z regułą po SNI, jak dla Traefika na stronie o równoważeniu.

3. Przejście k3s01 na etcd

Przed zmianą robimy kopię katalogu stanu, bo przejście z SQLite na etcd jest jednokierunkowe:

```
sudo systemctl stop k3s
sudo tar -czf /root/k3s-server-przed-etcd.tgz /var/lib/rancher/k3s/server
sudo systemctl start k3s
```

Tworzymy (albo uzupełniamy) plik konfiguracyjny `/etc/rancher/k3s/config.yaml`:

```
cluster-init: true
tls-san:
  - 10.20.10.69
  - k3s-api.ad.zielona.example
```

Wpis `cluster-init` każe k3s przy starcie zamienić wbudowaną bazę SQLite na etcd i zostać pierwszym członkiem klastra. Wpis `tls-san` dodaje adres wirtualny API i jego nazwę do certyfikatu apiserver, bo bez tego `kubectl` z adresu `10.20.10.69` odrzuci połączenie z powodu niezgodności certyfikatu. Restartujemy usługę i obserwujemy log:

```
sudo systemctl restart k3s
sudo journalctl -u k3s -f
```

W logu szukamy wpisów o etcd i o tym, że usługa wstała. Po restarcie sprawdzamy `kubectl get nodes` i kolorowy nginx: węzeł ma rolę `etcd`, a namespace `kolory` jest na miejscu. Jeśli zależy nam na czystym starcie, zamiast konwersji instalujemy k3s od nowa z `cluster-init` i stosujemy ponownie manifesty z `r7-01`.

> Zamiast konwersji możemy zaczynać od nowa: `sudo /usr/local/bin/k3s-uninstall.sh` na `k3s01`, ponowna instalacja i ponowne `kubectl apply -f`. Strata to tylko stan laboratorium, którego manifesty i tak mamy w plikach.

4. Dołączenie k3s02 i k3s03

Na `k3s01` odczytujemy token (karta „Odczyt tokenu klastra”). Na `k3s02` i `k3s03` zakładamy `config.yaml` z adresem pierwszego serwera, ze zmienną `TokenKlastra` zastąpioną odczytanym tokenem i z takim samym `tls-san` jak na `k3s01`:

```
sudo install -d -m 755 /etc/rancher/k3s
sudo tee /etc/rancher/k3s/config.yaml > /dev/null <<'EOF'
server: https://10.20.10.44:6443
token: TokenKlastra
tls-san:
  - 10.20.10.69
  - k3s-api.ad.zielona.example
EOF
sudo chmod 600 /etc/rancher/k3s/config.yaml
```

Instalujemy k3s w tej samej wersji co na `k3s01`, najpierw na `k3s02`, a po jego wejściu do klastra na `k3s03`:

```
curl -sfL https://get.k3s.io | INSTALL_K3S_VERSION='WersjaK3s' sh -s - server
```

Dołączamy węzły pojedynczo, bo przy dwóch jednoczesnych dołączeniach trudniej ustalić, który z nich zgłasza błąd. Po każdym dołączeniu wykonujemy `kubectl get nodes -o wide` i `kubectl get --raw='/readyz?verbose' | grep etcd`. Z czasem w kolumnie `ROLES` pojawia się `etcd` przy wszystkich trzech.

◻◻ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: wynik `kubectl get nodes -o wide` z trzema węzłami `Ready` i rolami `control-plane,etcd`.

5. Adres wirtualny API: keepalived na trzech serwerach

Na każdym z trzech serwerów instalujemy `keepalived`, ten sam pakiet co na `lb01` (komendy ze strony o równoważeniu). Skrypt kontrolny zapisujemy jako `/usr/local/sbin/ft-ha-sprawdz-k3s.sh` z uprawnieniami `0755` i właścicielem `root`:

```
#!/bin/bash
# /usr/local/sbin/ft-ha-sprawdz-k3s.sh
# Kod 0 tylko wtedy, gdy lokalny apiserver jest gotowy razem z etcd z kworum.
# k3s.yaml wskazuje https://127.0.0.1:6443 z poswiadczeniami administratora.
timeout 3 /usr/local/bin/k3s kubectl --kubeconfig /etc/rancher/k3s/k3s.yaml --request-
timeout=2s get --raw='/readyz' > /dev/null 2>&1
exit $?
```

Skrypt pyta o `/readyz` z poświadczeniami administratora k3s, a ta kontrola obejmuje etcd. Wcześniejsza wersja uznawała za zdrowy każdy apiserver, który odpowiadał kodem 200, 401 albo 403, czyli taki, który tylko żyje: po partycji sieci apiserver na odciętym serwerze odpowiada 401, choć nie ma etcd z kworum, więc keepalived trzymałby na nim adres, a druga strona partycji przejęłaby go także. Ten sam skrypt stosuje strona k3s w wysokiej dostępności: migawki etcd, wymiana węzła, odtworzenie i aktualizacja.

Plik `/etc/keepalived/keepalived.conf` na `k3s01`:

Pokaż pełny przykład (36 linii)

```
global_defs {
    router_id k3s01
    enable_script_security
    script_user root
}

vrrp_script ft_ha_sprawdz_k3s {
    script "/usr/local/sbin/ft-ha-sprawdz-k3s.sh"
    interval 3
    timeout 4
    fall 3
    rise 2
}

vrrp_instance VI_K3S_API {
    state BACKUP
    nopreempt
    interface ens192
    virtual_router_id 69
    priority 150
```

```
    advert_int 1

    unicast_src_ip 10.20.10.44
    unicast_peer {
        10.20.10.67
        10.20.10.68
    }

    virtual_ipaddress {
        10.20.10.69/24 dev ens192
    }

    track_script {
        ft_ha_sprawdz_k3s
    }
}
```

Na pozostałych serwerach zmieniamy te same pola co na parze `lb`:

Pole	k3s01	k3s02	k3s03
`router_id`	`k3s01`	`k3s02`	`k3s03`
`priority`	`150`	`120`	`100`
`unicast_src_ip`	`10.20.10.44`	`10.20.10.67`	`10.20.10.68`
`unicast_peer`	`.67` i `.68`	`.44` i `.68`	`.44` i `.67`

Identyfikator `virtual_router_id 69` jest inny niż 46 (para Dockera) i 66 (para `lb`), żeby trzy instancje VRRP nie myliły się w jednej sieci. Wszystkie trzy węzły startują w stanie `BACKUP` z `nopreempt`, więc adres, który już gdzieś działa, nie wraca na maszynę o wyższym priorytecie po jej powrocie. Skrypt nie ma wagi, więc przy trzech błędach z rzędu węzeł przechodzi w stan FAULT i oddaje adres.

Sprawdzamy składnię na każdym węźle, uruchamiamy usługę i szukamy adresu:

```
sudo keepalived --config-test
sudo systemctl enable --now keepalived
ip -br -4 addr show dev ens192
```

Adres `10.20.10.69/24` ma być tylko na jednym serwerze. Zaporę dla VRRP otwieraliśmy w kroku 2.

6. Kubeconfig na adresie wirtualnym i DNS

W DNS zakładamy rekord nazwy API (komenda z `dc01`):

```
Add-DnsServerResourceRecordA -ZoneName "ad.zielona.example" -Name "k3s-api" -IPv4Address
"10.20.10.69" -CreatePtr
```

Na stacji administratora w pliku kubeconfig zmieniamy pole `server` z `https://127.0.0.1:6443` (tak jest po skopiowaniu z `k3s01`) na `https://k3s-api.ad.zielona.example:6443` i sprawdzamy:

```
kubectl --kubeconfig ~/.kube/k3s-zielona.yaml get nodes
```

Gdy `kubectl` zgłasza błąd certyfikatu, oznacza to, że `tls-san` nie zawiera nazwy albo adresu: poprawiamy `config.yaml` na wszystkich trzech serwerach i restartujemy je po jednym (certyfikat apiserver odtwarza się przy starcie po zmianie listy SAN).

7. Longhorn

Na wszystkich trzech węzłach instalujemy wymagane pakiety i włączamy `iscsid`:

```
sudo dnf install -y iscsi-initiator-utils nfs-utils
sudo systemctl enable --now iscsid
```

Dysk danych (drugi wirtualny dysk, w tym laboratorium `/dev/sdb`) formatujemy i montujemy pod `/var/lib/longhorn`:

```
sudo mkfs.xfs /dev/sdb
sudo mkdir -p /var/lib/longhorn
echo '/dev/sdb /var/lib/longhorn xfs defaults 0 0' | sudo tee -a /etc/fstab
sudo mount -a
```

Instalacja Longhorna 1.13.0 to jedno zastosowanie manifestu z jego repozytorium. Przed nią sprawdzamy w dokumentacji Longhorna, czy wersja 1.13.0 wspiera Kubernetes 1.36, bo macierz zgodności zmienia się z wydaniami:

```
kubectl apply -f
https://raw.githubusercontent.com/longhorn/longhorn/v1.13.0/deploy/longhorn.yaml
kubectl -n longhorn-system get pods -w
```

Czekamy, aż wszystkie pody w `longhorn-system` będą `Running`. Potem zakładamy klasę z dwiema replikami (domyślna klasa `longhorn` ma ich trzy):

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: longhorn-2r
provisioner: driver.longhorn.io
allowVolumeExpansion: true
reclaimPolicy: Delete
volumeBindingMode: Immediate
parameters:
  numberOfReplicas: "2"
  staleReplicaTimeout: "30"
```

Zapisujemy ją jako `longhorn-2r.yaml` w `/docker/compose/k3s-kolory` i stosujemy `kubectl apply -f`. Do testu danych służy mały wolumen z podem, który zapisuje co 5 sekund datę do pliku:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dane-testowe
  namespace: kolory
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: longhorn-2r
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zapisywacz
  namespace: kolory
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: zapisywacz
```

```
  template:
    metadata:
      labels:
        app: zapisywacz
    spec:
      containers:
        - name: zapisywacz
          image: busybox:1.37.0
          command:
            - sh
            - -c
            - 'while true; do date >> /dane/czas.log; sleep 5; done'
          volumeMounts:
            - name: dane
              mountPath: /dane
      volumes:
        - name: dane
          persistentVolumeClaim:
            claimName: dane-testowe
```

Strategia `Recreate` jest tu konieczna, bo wolumen ReadWriteOnce może być zamontowany tylko na jednym węźle naraz, a domyślne `RollingUpdate` próbowałoby postawić nowy pod, zanim stary zwolni wolumen. Po zastosowaniu pliku sprawdzamy `kubectl -n longhorn-system get volumes.longhorn.io`: wolumen ma być `attached` i `healthy`, a po wpisie `kubectl -n kolory exec deploy/zapisywacz -- tail -n 2 /dane/czas.log` widzimy świeże daty.

◻◻ ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: ekran „Volumes” w interfejsie Longhorna z jednym wolumenem `healthy` i dwiema replikami na różnych węzłach (dostęp przez `kubectl -n longhorn-system port-forward svc/longhorn-frontend 8000:80`).

8. Kolorowy nginx w wersji z trzema węzłami i HAProxy

Bierzemy manifest z `r7-01` i zmieniamy trzy rzeczy: liczbę replik na sześć, rozkład podów po węzłach i dodajemy PodDisruptionBudget, żeby `drain` nie zabierał naraz więcej niż dwóch podów. W `Deployment` zmieniamy:

```
spec:
  replicas: 6
  template:
    spec:
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: kubernetes.io/hostname
          whenUnsatisfiable: ScheduleAnyway
          labelSelector:
            matchLabels:
              app: kolorowy-nginx
```

Na końcu pliku dopisujemy:

```
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: kolorowy-nginx
  namespace: kolory
spec:
  maxUnavailable: 2
  selector:
    matchLabels:
      app: kolorowy-nginx
```

Reguła `topologySpreadConstraints` z `ScheduleAnyway` dąży do równego rozkładu po trzech węzłach (po dwa pody na węzeł), ale w razie braku miejsca nie blokuje startu poda. Stosujemy `kubectl apply -f kolorowy-nginx.yaml` i sprawdzamy rozkład przez `kubectl -n kolory get pods -o wide`.

Do `haproxy.cfg` na `lb01` i `lb02` dopisujemy w `fe_http` pod linią `use_backend be_mailcow_http` regułę i dodajemy backend:

```
    acl host_kolory hdr(host),field(1,:) -i kolory.zielona.example
    use_backend be_k3s_http if host_kolory
```

```
backend be_k3s_http
    mode http
    balance roundrobin
    option httpchk
    http-check send meth GET uri / ver HTTP/1.1 hdr Host kolory.zielona.example
```

```
    http-check expect rstatus ^[2-4][0-9][0-9]$
    default-server inter 3s fall 3 rise 2
    server k3s01 10.20.10.44:80 check
    server k3s02 10.20.10.67:80 check
    server k3s03 10.20.10.68:80 check
```

Po `haproxy -c -f /etc/haproxy/haproxy.cfg` i `systemctl reload haproxy` na obu maszynach `lb` sprawdzamy drogę żądania. Jeżeli to samo `kolory.zielona.example` było wcześniej kierowane na Swarm (backend `be_swarm_kolory`), usuwamy tamtą regułę, bo oba laboratoria stawiamy po kolei.

Weryfikacja

```
kubectl get nodes -o wide
kubectl get --raw='/readyz?verbose' | grep etcd
ip -br -4 addr show dev ens192
kubectl -n kolory get pods -o wide
kubectl -n longhorn-system get volumes.longhorn.io
```

Oczekujemy trzech węzłów `Ready`, linii `[+]etcd ok`, adresu `10.20.10.69` na dokładnie jednym serwerze, sześciu podów `Running` rozłożonych po dwa na węzeł i wolumenu `attached` oraz `healthy`. Ruch z zewnątrz sprawdzamy pętlą, tak jak na stronie o pierwszym wdrożeniu, ale przez adres wirtualny równoważenia:

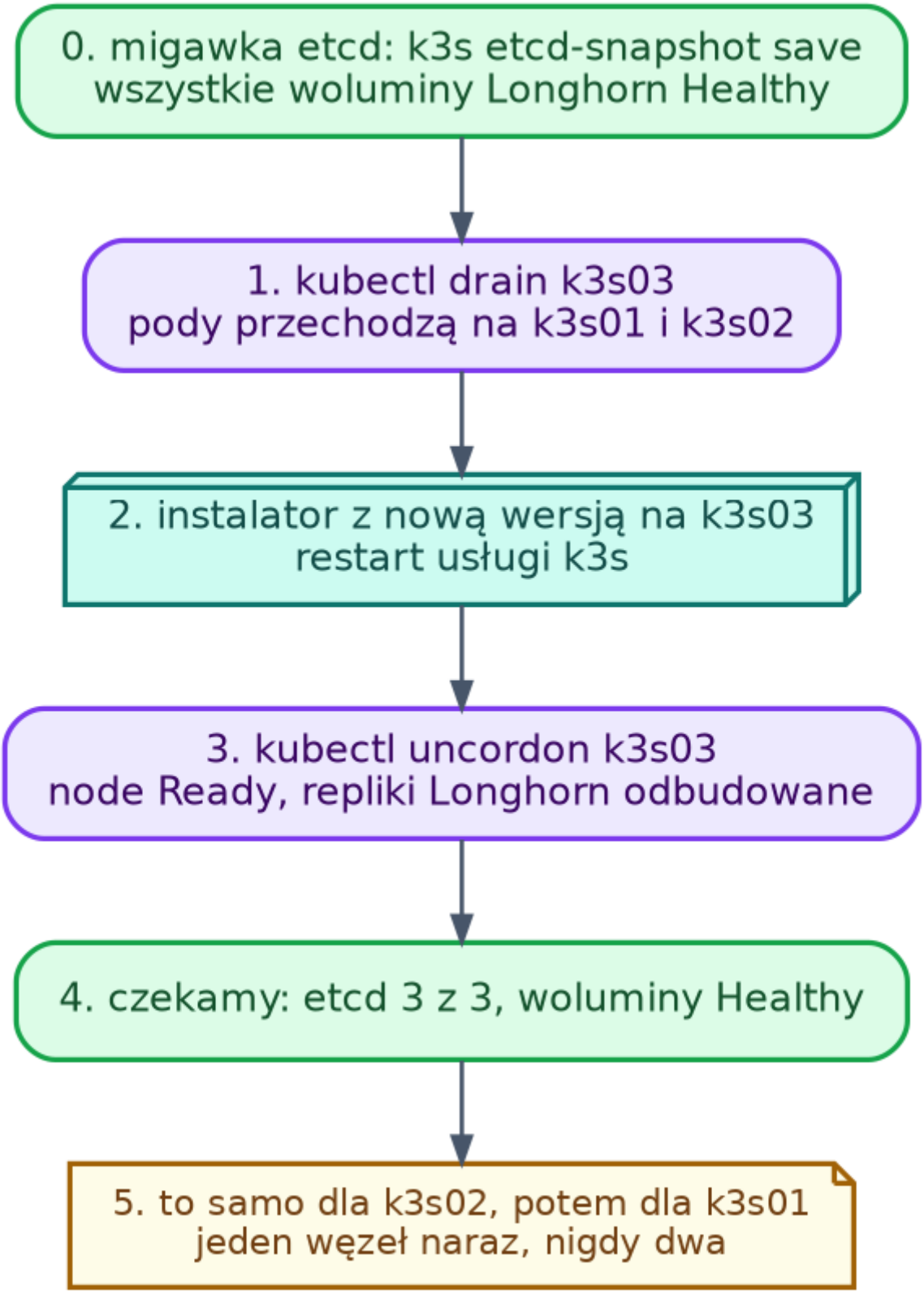
```
for i in $(seq 1 12); do
  curl -s --resolve kolory.zielona.example:80:10.20.10.66 http://kolory.zielona.example/ |
grep "Kolor:"
done
```

Wynik zawiera różne kolory, a w statystykach HAProxy (port 8404 na `lb01`) wszystkie trzy serwery `be_k3s_http` są `UP`. Adres API sprawdzamy osobno:

```
curl -sk -o /dev/null -w '%{http_code}\n' https://10.20.10.69:6443/readyz
```

Aktualizacja węzłów po kolei

Najczęstszy powód wyłączania węzła to aktualizacja, więc jest to najlepszy test całego układu. Zasada jest taka: jeden węzeł naraz, a między węzłami czekamy, aż klaster wróci do pełnego zdrowia.



Rysunek 3. Cykl aktualizacji jednego węzła: migawka, drain, nowa wersja, uncordon i czekanie na zdrowe etcd i woluminy, potem następny węzeł (kliknięcie otwiera pełny rozmiar)

Przed pierwszym węzłem robimy migawkę etcd (karta „Zapis migawki etcd”) i sprawdzamy, że wszystkie wolumeny Longhorna są `healthy`. Potem dla każdego węzła w kolejności `k3s03`, `k3s02`, `k3s01`:

1. `kubectl drain NazwaWezla --ignore-daemonsets --delete-emptydir-data`. Jeśli wisi, sprawdzamy `kubectl get pdb -A`, bo budżet zakłóceń może nie pozwalać na wyrzucenie kolejnego poda.
2. Na węźle: `curl -sfL https://get.k3s.io | INSTALL_K3S_VERSION='NowaWersjaK3s' sh -s - server`. Instalator czyta istniejący `config.yaml`.
3. `kubectl uncordon NazwaWezla` po tym, jak węzeł jest `Ready` w nowej wersji.
4. Czekamy na `[+]etcd ok`, trzy węzły `Ready` i na wszystkie wolumeny `healthy`. Longhorn po powrocie węzła odbudowuje repliki i w tym czasie kondycja wolumenów jest `degraded`.

Adres wirtualny API zmienia właściciela, gdy aktualizowany węzeł go trzymał, więc `kubectl` na chwilę zgłasza błąd. W odczytach zawsze używamy nazwy `k3s-api`, nie adresu serwera. Wersje nie mogą przeskakiwać pomiędzy wydaniami podrzędnymi Kubernetesa (z 1.36 na 1.37, nie na 1.38), a serwery aktualizujemy przed agentami, o ile jakieś mamy. Nigdy nie wyłączamy kolejnego węzła, dopóki poprzedni nie wrócił do pełnej gotowości, bo wtedy tracimy kworum w środku aktualizacji.

Test awarii

Wszystkie testy robimy z pętlą `curl` z sekcji „Weryfikacja” w drugim oknie. Czas przerwy zapisujemy w tabeli wyników. Nie robimy testów w okresie zakazu zmian od 15 listopada do 6 stycznia i nie w piątek po południu.

A. Planowe wyłączenie węzła

1. `kubectl drain k3s03 --ignore-daemonsets --delete-emptydir-data`. Sprawdzamy `kubectl -n kolory get pods -o wide`: pody z `k3s03` przechodzą na `k3s01` i `k3s02`, a pętla `curl` nie widzi przerwy.
2. Na `k3s03` wykonujemy `sudo systemctl poweroff`. HAProxy po kilku kontrolach oznacza `k3s03` jako `DOWN`. Wolumen Longhorna z repliką na `k3s03` przechodzi w stan `degraded`.
3. Uruchamiamy `k3s03` ponownie i wykonujemy `kubectl uncordon k3s03`. Obserwujemy powrót wolumenu do `healthy`.

B. Twarda awaria jednego węzła (jak awaria maszyny)

1. Na `hv02`, który jest właścicielem roli: `Stop-VM -Name k3s02 -ComputerName hv02 -TurnOff -Force` (maszyna traci zasilanie bez zamykania systemu). Nie używamy `Stop-ClusterGroup`,

bo wyłącza on rolę według akcji ustawionej w klastrze, która może zapisać stan maszyny zamiast ją wyłączyć (do potwierdzenia w ustawieniach roli), a to nie jest twarda awaria.
2. HAProxy wyłącza węzeł z rotacji po trzech nieudanych kontrolach, czyli po około 9 sekundach. Zgodnie z domyślnym ustawieniem kontrolera węzłów (`--node-monitor-grace-period`, według dokumentacji kube-controller-manager 50 sekund [14]) węzeł przechodzi w `NotReady`, a jego pody tracą gotowość. Dopiero po domyślnych 300 sekundach (`tolerationSeconds` taintu) pody są usuwane i odtwarzane na pozostałych węzłach, więc przez około 5 minut pracujemy z mniejszą liczbą kopii.
3. Spodziewamy się, że przez okres do około 50 sekund Traefik na pozostałych węzłach może jeszcze kierować część żądań na pody z `k3s02`, bo ich adresy są na liście endpointów, dopóki węzeł nie zostanie uznany za niedziałający. To zjawisko mierzymy i wpisujemy do wyników, nie zakładamy z góry, że go nie będzie.
4. `Start-VM -Name k3s02 -ComputerName hv02` kończy test.

C. Awaria hosta (utrata kworum)

1. Symulujemy utratę `hv01` zatrzymaniem dwóch maszyn z hosta: `Stop-VM -Name k3s01 -ComputerName hv01 -TurnOff -Force` i `Stop-VM -Name k3s03 -ComputerName hv01 -TurnOff -Force`.
2. `kubectl get nodes` przez `k3s-api` kończy się błędem albo przekroczeniem czasu: etcd bez kworum nie przyjmuje zapisów. Pody na `k3s02` dalej pracują, więc przez `lb` strona nadal odpowiada, ale nie możemy niczego zmienić, a lista endpointów jest zamrożona.
3. Uruchamiamy jedną z maszyn (`Start-VM -Name k3s01 -ComputerName hv01`). Po jej starcie kworum wraca (2 z 3) i klaster ożywa. W prawdziwej awarii hosta robi to klaster Hyper-V, który uruchamia maszyny na `hv02` samodzielnie, a maszyn wyłączonych przez `Stop-VM` sam nie uruchamia.
4. Czas od zatrzymania do pierwszej udanej komendy `kubectl` zapisujemy jako czas przywrócenia kworum.

D. Awaria adresu wirtualnego API

1. Na serwerze, który trzyma `10.20.10.69` (patrz `ip -br -4 addr`), zatrzymujemy usługę: `sudo systemctl stop k3s`. Zatrzymanie usługi nie kończy działających podów, więc dla pełnego efektu używamy `sudo k3s-killall.sh`.
2. Po trzech nieudanych kontrolach, czyli po około 9 sekundach, keepalived zmienia stan tego serwera na FAULT, a adres przechodzi na serwer o wyższym priorytecie z pozostałych.
3. Polecenie `kubectl --kubeconfig ~/.kube/k3s-zielona.yaml get nodes` zaczyna znowu odpowiadać i pokazuje zatrzymany węzeł jako `NotReady`.
4. `sudo systemctl start k3s` na zatrzymanym serwerze. Adres zostaje tam, gdzie jest (nopreempt), więc przeniesienie go z powrotem robimy ręcznie, o ile zależy nam na pierwotnym układzie.

Co sprawdziliśmy w piaskownicy

W piaskownicy nie było trzech maszyn, hostów Hyper-V ani samego k3s, więc nie sprawdziliśmy ani etcd, ani Longhorna, ani zachowania `kubectl` po awarii. Sprawdziliśmy to, co dało się sprawdzić bez nich.

Co	Wynik
`keepalived --config-test` na pliku `VI_K3S_API` z tej strony	składnia poprawna (interfejs w przestrzeni nazw sieci nazwany `ens192`)
Skrypt `ft-ha-sprawdz-k3s.sh`	`bash -n` bez błędów
`haproxy -c -f` na pełnej konfiguracji z `be_k3s_http` i blokiem `listen k3s_api`	konfiguracja poprawna
Backend `be_k3s_http` z kontrolą z nagłówkiem `Host` na trzech zastępczych serwerach HTTP (HAProxy 2.8)	wszystkie `UP`, ruch rozkłada się na trzy; po zatrzymaniu jednego serwera i 11 sekundach ruch idzie na dwa pozostałe, a ten serwer ma stan `DOWN`
Pliki YAML z tej strony (config k3s, StorageClass, PVC i Deployment, PDB)	parsują się jako poprawny YAML

Czasy z testów A do D, zachowanie endpointów po awarii węzła i odbudowa replik Longhorna pozostają do zmierzenia w laboratorium. Wpisujemy je do tabeli wyników, a nie szacujemy.

Wariant: API przez parę lb

Jeśli nie chcemy trzymać keepalived na serwerach k3s, adres API może obsługiwać para `lb`. W `haproxy.cfg` na `lb01` i `lb02` dopisujemy blok i zmieniamy `server` w kubeconfig na `https://lb-ha.ad.zielona.example:6443` (adres `10.20.10.66`, rekord z strony o równoważeniu):

```
listen k3s_api
    bind :6443
    mode tcp
    option tcplog
    option tcp-check
    default-server inter 3s fall 3 rise 2
    server k3s01 10.20.10.44:6443 check
    server k3s02 10.20.10.67:6443 check
    server k3s03 10.20.10.68:6443 check
```

`tls-san` musi wtedy zawierać `10.20.10.66` i `lb-ha.ad.zielona.example`. Zaletą jest rozkładanie ruchu na trzy serwery. Wadą jest to, że dostęp do klastra zależy od pary `lb`, a ta zależy od klastra

Hyper-V, więc wybieramy ten wariant tylko wtedy, gdy para `lb` i tak jest u nas ważniejszym elementem niż serwery k3s.

Kopia i odtworzenie

Migawki etcd k3s zapisuje domyślnie co 12 godzin i trzyma ostatnich pięć w `/var/lib/rancher/k3s/server/db/snapshots`. Domyślne wartości sprawdzamy w dokumentacji `k3s etcd-snapshot` w dniu wdrożenia. Migawki z jednego z trzech serwerów kopiujemy razem z tokenem na `bkp01` (restic, jak pozostałe dane) i sprawdzamy odtworzenie raz na kwartał w laboratorium.

Odtworzenie z migawki wykonujemy na jednym serwerze, po zatrzymaniu usługi `k3s` na wszystkich trzech:

```
sudo systemctl stop k3s
sudo k3s server --cluster-reset --cluster-reset-restore-
path=/var/lib/rancher/k3s/server/db/snapshots/NazwaMigawki
```

Pozostałe dwa serwery po resetowaniu usuwamy z klastra i dołączamy od nowa, kasując ich katalog `/var/lib/rancher/k3s/server/db`. Pełną procedurę czytamy z dokumentacji k3s przed użyciem, bo jest destrukcyjna.

Wycofanie zmian

```
kubectl delete namespace kolory
kubectl delete -f
https://raw.githubusercontent.com/longhorn/longhorn/v1.13.0/deploy/longhorn.yaml
```

Longhorna usuwamy dopiero po skasowaniu wszystkich PVC, które z niego korzystają. Węzły `k3s03` i `k3s02` usuwamy z klastra tak: `kubectl drain`, `kubectl delete node NazwaWezla` i na samym węźle `sudo /usr/local/bin/k3s-uninstall.sh`. Gdy chcemy wrócić do jednego węzła, robimy świeżą instalację `k3s01` bez `cluster-init`, bo powrotu z etcd do SQLite nie ma.

Z `haproxy.cfg` usuwamy backend `be_k3s_http` i regułę w `fe_http`, po czym `haproxy -c` i `reload`. Na każdym z trzech węzłów `sudo systemctl disable --now keepalived`, usuwamy plik `/etc/keepalived/keepalived.conf` i skrypt `ft-ha-sprawdz-k3s.sh`, a rekord `k3s-api` kasujemy w DNS. Strefę zapory `k3s` usuwa `sudo firewall-cmd --permanent --delete-zone=k3s` i `--reload`. Maszyny `k3s02` i `k3s03` kasujemy w klastrze Hyper-V, jeśli nie są już potrzebne, a adresy `10.20.10.67` do `10.20.10.69` wracają do puli.

Częste błędy

Objaw	Przyczyna	Co robimy
`k3s02` nie dołącza, w logu błąd tokenu	zły token albo `config.yaml` z błędem wcięcia	token z `k3s01`, `sudo journalctl -u k3s`, składnia YAML
Dołączony węzeł ma inną wersję niż pierwszy	instalator pobrał najnowszą wersję	zawsze `INSTALL_K3S_VERSION` równe wersji z `k3s --version` na `k3s01`
`kubectl` na `10.20.10.69` zgłasza błąd certyfikatu	brak adresu lub nazwy w `tls-san`	uzupełniamy `config.yaml` na wszystkich serwerach i restartujemy je po jednym
Po wyłączeniu dwóch węzłów `kubectl` nie odpowiada	utrata kworum etcd	uruchamiamy jeden z wyłączonych węzłów, nie kasujemy danych etcd
`kubectl drain` wisi	PodDisruptionBudget albo pod bez kontrolera	`kubectl get pdb -A`, decyzja o `--force` tylko dla podów bez kontrolera
Wolumen Longhorna `degraded` po powrocie węzła	trwa odbudowa repliki	czekamy, `kubectl -n longhorn-system get volumes.longhorn.io`, nie ruszamy kolejnego węzła
Adres API skacze między serwerami	skrypt kontrolny zgłasza błąd na zdrowym serwerze	uruchamiamy skrypt ręcznie i sprawdzamy `k3s kubectl get --raw='/readyz?verbose'`, która kontrola zawodzi (często etcd na wolnym dysku)
Pod `zapisywacz` nie startuje po przeniesieniu	wolumen RWO nadal zamontowany na starym węźle	`Recreate` w Deployment, `kubectl describe pod`, czekamy na odmontowanie
Po awarii węzła strona przez chwilę zwraca błędy	endpointy zawierają pody z niedziałającego węzła	to oczekiwane do czasu `NotReady` (około 50 s), mierzymy i dokumentujemy
Trzy repliki Longhorna: wolumen nie odbudowuje się	trzy repliki na trzech węzłach, a jeden węzeł padł	klasa z dwiema replikami albo czwarty węzeł

□□ Ćwiczenia

1. Dlaczego trzy węzły etcd na dwóch hostach Hyper-V nie chronią przed awarią hosta, a chronią przed awarią maszyny?

Odpowiedź

Kworum to co najmniej 2 z 3 głosów. Awaria maszyny zabiera jeden głos, więc zostają dwa. Awaria hosta, na którym stoją dwa węzły, zabiera dwa głosy naraz i zostaje jeden. Tę awarię nadal pokrywa klaster Hyper-V, który po kilku minutach uruchamia maszyny na drugim hoście.

1. Dlaczego wybieramy dwie repliki Longhorna, a nie trzy, skoro domyślna wartość to trzy?

Odpowiedź

S2D z dwukierunkowym lustrem już trzyma dwie kopie bloku, więc trzy repliki dają łącznie sześć kopii i duży narzut zapisu. Trzy repliki na trzech węzłach nie mają też wolnego węzła na odbudowę po awarii jednego z nich, bo Longhorn domyślnie nie kładzie dwóch replik wolumenu na tym samym węźle. Dwie repliki chronią przed awarią węzła k3s, a przed awarią dysku chroni S2D.

1. Po aktualizacji `k3s03` wszystkie trzy węzły są `Ready`. Czemu mimo to nie aktualizujemy od razu `k3s02`?

Odpowiedź

Longhorn po powrocie węzła odbudowuje repliki i wolumeny są przez ten czas `degraded`. Gdyby w tym czasie wyłączyć kolejny węzeł, wolumen mógłby stracić drugą replikę, a etcd po restarcie `k3s03` może jeszcze się synchronizować. Czekamy na `[+]etcd ok` i `healthy` przy wszystkich wolumenach.

1. W teście C `kubectl` przestaje odpowiadać, a strona dalej działa. Dlaczego, i co jest wtedy niedostępne?

Odpowiedź

Bez kworum etcd nie przyjmuje zapisów, więc API nie może niczego zmienić. Pody na `k3s02` i Traefik na nim działają dalej i obsługują ruch na podstawie ostatniego znanego stanu. Niedostępne są wszystkie zmiany: skalowanie, wdrożenia, a także samonaprawa, bo kontrolery potrzebują API.

Powiązane: Wysoka dostępność: usługa, która nie ma jednego serwera · Wysoka dostępność: pojęcia, liczenie dostępności i poziomy rezerwy · Przenoszenie kontenerów z docker01 na docker02: zimna i ciepła rezerwa, przełączenie · Replikacja danych między hostami: DRBD, GlusterFS, Ceph, ZFS i replikacja w aplikacji · Równoważenie i adres wirtualny: keepalived i HAProxy na parze lb01 i lb02 · Docker Swarm: klaster z trzema menedżerami, na którym

przenosimy kolorowy nginx · k3s w laboratorium: pierwszy deploy, nginx w kolorach · Hyper-V: maszyny wirtualne, punkty kontrolne i migracja na żywo · k3s w wysokiej dostępności: migawki etcd, wymiana węzła, odtworzenie i aktualizacja

Źródła:

1. docs.k3s.io: High Availability Embedded etcd
2. docs.k3s.io: Cluster Load Balancer
3. docs.k3s.io: Requirements
4. docs.k3s.io: Backup and Restore
5. docs.k3s.io: Manual Upgrades
6. docs.k3s.io: v1.36.X release notes i v1.37.X release notes (v1.36.5+k3s1 i v1.37.1+k3s1 z 30 września 2026, odczyt 18 października 2026)
7. longhorn.io: Install with Kubectl
8. longhorn.io: Best Practices
9. longhorn.io: Settings
10. kube-vip.io: Documentation
11. etcd.io: FAQ (liczba członków i tolerancja awarii)
12. kubernetes.io: Safely Drain a Node
13. learn.microsoft.com: Set-ClusterOwnerNode
14. kubernetes.io: kube-controller-manager (`--node-monitor-grace-period` domyślnie 50 s, odczyt 18 października 2026)

Revision #16
Created 2026-10-03 13:44:11 UTC by Tomasz Konieczny
Updated 2026-10-26 14:21:54 UTC by Tomasz Konieczny