Docker Swarm: klaster z trzema menedżerami, na którym przenosimy kolorowy nginx

Poziom: średnio

Dotyczy: Docker Engine 29 (tryb Swarm), Rocky Linux 9, HAProxy z poprzedniej strony, laboratorium na trzech maszynach swarm01 do swarm03

Czas: ~90 min

Swarm to tryb pracy Dockera, w którym kilka maszyn tworzy jeden klaster i samo pilnuje, żeby usługa miała tyle działających kopii, ile zadeklarowaliśmy. Na tej stronie budujemy go w laboratorium z trzech maszyn i przenosimy na niego kolorowy nginx ze strony o k3s. Na końcu oceniamy uczciwie, kiedy Swarm ma sens dla Zielonej, a kiedy nie.

ZAŁOŻENIA SCENARIUSZA

laboratorium to trzy maszyny `swarm01` do `swarm03` na dwóch hostach `hv01` i `hv02`, w VLAN 10 za parą HAProxy z poprzedniej strony. Dwa hosty fizyczne znaczą, że jeden z nich trzyma dwie maszyny, więc nie jest to pełna odporność na awarię hosta.

Skróty użyte na tej stronie (12)

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
ESP	Enrollment Status Page: strona postępu podczas pierwszej konfiguracji, która trzyma użytkownika na ekranie, dopóki Intune nie zainstaluje wskazanych aplikacji i profili

Skrót	Co znaczy
etcd	etc distributed: rozproszony magazyn par klucz-wartość, w którym Kubernetes trzyma cały stan klastra, a zmiany uzgadnia protokołem Raft
HAProxy	High Availability Proxy: program, który rozdziela połączenia między kilka serwerów i omija te, które nie odpowiadają
HTTP	Hypertext Transfer Protocol: protokół stron WWW
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
Raft	protokół uzgadniania stanu w klastrze: zmiana jest przyjęta, gdy zgodzi się większość węzłów; na nim stoją menedżerowie Swarm i etcd
UDP	User Datagram Protocol: szybki protokół sieciowy bez potwierdzeń
VLAN	Virtual LAN: wydzielona logicznie sieć na tym samym sprzęcie
VXLAN	Virtual Extensible LAN: technika tunelowania ruchu sieciowego w pakietach UDP (port 8472 w k3s), którą sieć podów przechodzi między węzłami

Fiszka

Komendy wykonujemy z CLI bash z uprawnieniami root (sudo) albo jako user z grupy `docker`, na maszynie `swarm01` (pierwszy menedżer), chyba że przy komendzie napisano inaczej. Adresy z laboratorium: `swarm01` to `10.20.10.67`, `swarm02` to `10.20.10.68`, `swarm03` to `10.20.10.69`, adres wirtualny równoważenia to `10.20.10.66`. Bloki „Przykład” pokazują kształt wyniku, nie pochodzą z prób na trzech maszynach Zielonej (opis tego, co sprawdziliśmy, jest w sekcji „Co sprawdziliśmy w piaskownicy”).

□□ ZMIENIA SYSTEM

Założenie klastra na pierwszym menedżerze: docker swarm init

```
docker swarm init --advertise-addr 10.20.10.67
```

Maszyna zostaje pierwszym menedżerem i liderem Raft, a Docker wypisuje polecenie dołączenia dla pracowników. Adres w `--advertise-addr` musi być stały, bo inne węzły łączą się z nim po restarcie.

Przykład:

```
Swarm initialized: current node (7g2u4l4r7cb46781es8nrmynp) is now a manager.
```

```
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-... 10.20.10.67:2377
```

Raft = protokół uzgadniania stanu w klastrze: zmiana jest przyjęta, gdy zgodzi się większość węzłów; na nim stoją menedżerowie Swarm i etcd · **docker** = narzędzie do kontenerów

ZMIENIA SYSTEM

Odczyt polecenia dołączenia menedżera: docker swarm join-token manager

```
docker swarm join-token manager
```

Wypisuje komendę z tokenem dla menedżera. Token menedżera daje pełną władzę nad klastrem, więc traktujemy go jak hasło.

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Dołączenie węzła do klastra: docker swarm join

```
docker swarm join --token TokenMenedzera 10.20.10.67:2377
```

> (gdzie `TokenMenedzera` to token z poprzedniej karty; wykonujemy to na `swarm02` i `swarm03`)

docker = narzędzie do kontenerów

TYLKO ODCZYT

Lista węzłów klastra: docker node ls

```
docker node ls
```

Pokazuje wszystkie węzły, ich stan i rolę. W kolumnie `MANAGER STATUS` stoi `Leader` przy jednym węźle i `Reachable` przy pozostałych menedżerach.

Przykład:

```
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS
ENGINE VERSION
a1b2c3d4e5f6g7h8i9j0k1l2m *   swarm01    Ready     Active         Leader
29.8.2
b2c3d4e5f6g7h8i9j0k1l2m3n     swarm02    Ready     Active         Reachable
29.8.2
c3d4e5f6g7h8i9j0k1l2m3n4o     swarm03    Ready     Active         Reachable
29.8.2
```

docker = narzędzie do kontenerów

TYLKO ODCZYT

Wdrożenie stosu z pliku: docker stack deploy

```
docker stack deploy -c /docker/compose/kolory-swarm/kolory.yaml kolory
```

Tworzy usługi, sieci i konfiguracje opisane w pliku, a przy kolejnym uruchomieniu z innym plikiem aktualizuje tylko to, co się zmieniło. Ostatni argument to nazwa stosu, która staje się przedrostkiem nazw (`kolory_kolorowy-nginx`).

docker = narzędzie do kontenerów

TYLKO ODCZYT

Sprawdzenie, gdzie pracują zadania: docker service ps

```
docker service ps kolory_kolorowy-nginx --format '{{.Name}} {{.Node}} {{.CurrentState}}'
```

Pokazuje każde zadanie, węzeł, na którym stoi, i jego stan. Szukamy stanów `Running` i równomiernego rozkładu po węzłach.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Zmiana liczby kopii usługi: docker service scale

```
docker service scale kolory_kolorowy-nginx=6
```

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Aktualizacja obrazu usługi: docker service update --image

```
docker service update --image nginx:NowaWersja kolory_kolorowy-nginx
```

Wymienia zadania po kolei według zasad `update_config` z pliku stosu. Ta sama komenda z opcją `--force` bez zmiany obrazu robi stopniowy restart wszystkich zadań.

docker = narzędzie do kontenerów

TYLKO ODCZYT

Wycofanie ostatniej aktualizacji usługi: docker service rollback

```
docker service rollback kolory_kolorowy-nginx
```

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Opróżnienie węzła przed pracami: docker node update --availability drain

```
docker node update --availability drain swarm03
```

Zadania z `swarm03` przechodzą na pozostałe węzły, a nowe tam nie trafiają. Powrót: ta sama komenda z `active`. Po powrocie Swarm nie przenosi zadań z powrotem sam.

docker = narzędzie do kontenerów

ZMIENIA SYSTEM

Utworzenie sekretu z wpisanego hasła: docker secret create

```
read -r -s -p "Hasło: " HASLO; printf '%s' "${HASLO}" | docker secret create NazwaSekretu -; unset HASLO
```

> (gdzie `NazwaSekretu` to nazwa, pod którą usługa odczyta wartość jako plik `/run/secrets/NazwaSekretu`; wpisanie hasła w samej komendzie zostawiłoby je w historii powłoki)

read = bash: wczytaj wiersz z klawiatury do zmiennej, a z opcją -s bez wypisywania wpisanego tekstu na ekran, na przykład dla tokenu API

OSTROŻNIE · NIEODWRACALNE

Odzyskanie klastra po utracie kworum: docker swarm init --force-new-cluster

```
docker swarm init --force-new-cluster --advertise-addr 10.20.10.67
```

> Ostatnia deska ratunku, gdy utracone menedżerowie nie wrócą. Maszyna, na której to wykonujemy, zostaje jedynym menedżerem klastra z własnej kopii stanu, więc zmiany z czasu rozdzielenia znikają, a pozostałe węzły dołączamy od nowa. Wcześniej próbujemy uruchomić brakujące maszyny.

docker = narzędzie do kontenerów

OSTROŻNIE · NIEODWRACALNE

Usunięcie stosu: docker stack rm

```
docker stack rm kolory
```

Usuwa usługi i sieci stosu. Konfiguracje z nazwą nadaną jawnie (`name:` w pliku) i sekrety `external` zostają, bo stos ich nie tworzył.

docker = narzędzie do kontenerów

Z życia wzięte

Z ŻYCIA WZIĘTE

Czwartek, 10:20. Tomek stawia trzy maszyny na `hv01` i `hv02`, żeby pokazać Oli, jak wygląda „prawdziwy klaster”. Przepisuje plik Compose z `docker01` na stos i dostaje ostrzeżenie, że `container_name` i `restart` są ignorowane, a bind mount `/docker/apps/gitea` kończy się błędem na dwóch maszynach, na których taki katalog nie istnieje. Ola mówi, że to właśnie jest odpowiedź na pytanie pana Zbigniewa, czemu nie przenosimy wszystkiego od razu: klaster rozwiązuje kopie aplikacji, a nie ich dane.

Jak to działa

Rozwiń: jak to działa pod spodem

Stan projektu w 2026 roku

Pierwszy Swarm, dziś nazywany Swarm Classic, powstał w 2014 roku jako osobny projekt, który przez pośrednika API łączył kilka hostów Dockera w jeden wirtualny host; jego repozytorium zarchiwizowano 1 lutego 2021 roku [12]. Tryb Swarm, którego używamy na tej stronie, to co innego: wszedł do samego silnika w Dockerze 1.12, wydanym 28 lipca 2016 roku, i jest zbudowany na bibliotece SwarmKit [12][13].

Biznes Docker Enterprise przejął Mirantis w 2019 roku. 1 lipca 2025 roku Mirantis ogłosił, że Swarm będzie w pełni wspierany „przez co najmniej 2030” jako część Mirantis Kubernetes Engine 3 [1]. To deklaracja o produkcie komercyjnym, nie o samym silniku Dockera.

Docker Engine w wersji 29 (najnowsze wydanie 29.8.2 z 30 września 2026) nadal ma w dokumentacji komplet stron o trybie Swarm [2]. Według obserwacji z marca 2026 roku Swarm działa, ale od dawna nie dostaje dużych nowości [3]. Tę ocenę przyjmujemy z blogu, nie z oficjalnego dokumentu projektu, więc czytelnik powinien sprawdzić ją sam w repozytorium `moby/swarmkit`.

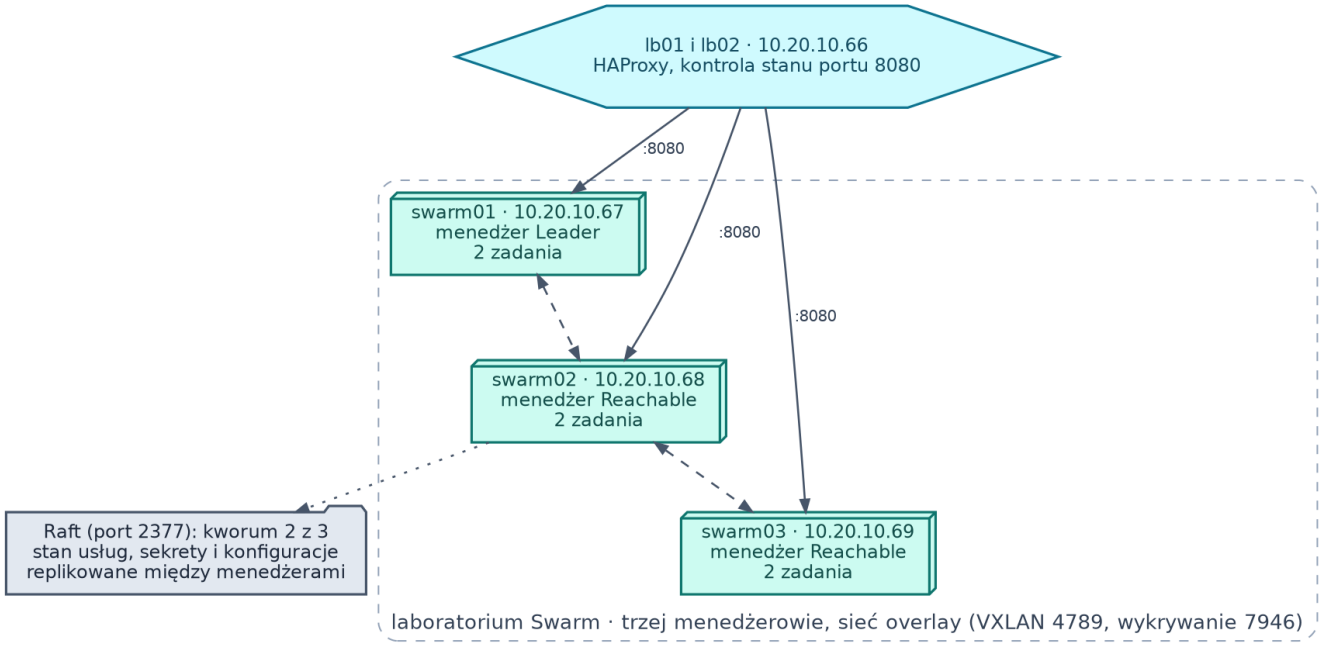
Dla Zielonej wynika z tego tyle: nie ma ryzyka, że Swarm zniknie w przyszłym roku, ale też nie ma obietnicy rozwoju. Wybór między Swarmem a Kubernetesem robimy więc na podstawie tego, czego nam brakuje dziś, a nie na podstawie reputacji.

Z czego składa się klaster

Pojęcie: Swarm. Tryb Dockera, w którym kilka maszyn z Dockerem (węzłów) tworzy jeden klaster. Administrator opisuje usługę (obraz, liczba kopii, porty), a Swarm rozmieszcza kontenery na węzłach i odtwarza je, gdy któryś padnie.

Pojęcie: usługa i zadanie. Usługa (service) to opis tego, co ma działać: obraz, liczba kopii, porty. Zadanie (task) to jedna działająca kopia, czyli jeden kontener na jednym węźle. Gdy zadanie pada, Swarm tworzy nowe, już pod inną nazwą.

Pojęcie: stos (stack). Zestaw usług, sieci i konfiguracji opisany jednym plikiem i wdrażany jedną komendą. To odpowiednik katalogu Compose na jednym hoście, tylko rozłożony po klastrze.



Rysunek 1. Laboratorium Swarm: trzej menedżerowie, protokół Raft między nimi, sieć overlay po VXLAN i HAProxy z parą lb01 i lb02 przed klastrem. Każdy węzeł prowadzi dwa zadania kolorowego nginksa. (kliknięcie otwiera pełny rozmiar)

Węzły mają dwie role. Menedżer przechowuje stan klastra i podejmuje decyzje, a pracownik tylko uruchamia zadania. Domyślnie menedżer także uruchamia zadania [4], co w trzywęzłowym laboratorium jest potrzebne, bo nie mamy osobnych pracowników. W większych klastrach menedżerów można opróżnić (`drain`), żeby pracowali tylko jako sterowanie.

Kworum i protokół Raft

Menedżerowie uzgadniają stan protokołem Raft: zmiana jest przyjęta, gdy zgodzi się większość, czyli kworum. Mechanizm jest ten sam, który opisaliśmy w teorii na stronie Wysoka dostępność: pojęcia, liczenie dostępności i poziomy rezerwy, a w k3s działa jako etcd.

Liczba menedżerów	Potrzebna większość	Tolerowane awarie
1	1	0
3	2	1
5	3	2
7	4	3

Dokumentacja zaleca nieparzystą liczbę menedżerów i nie więcej niż siedmiu, bo większa liczba nie poprawia wydajności, tylko ją pogarsza [5]. Trzej menedżerowie to minimum, które przeżyje awarię jednego z nich. Dwóch menedżerów jest gorsze niż jeden, bo awaria któregokolwiek zatrzymuje klaster.

Co się dzieje po utracie kworum, wynika z dokumentacji [4]. Działające zadania na węzłach działają dalej, ale nie da się niczego dodać, zmienić ani uruchomić, bo nikt nie podejmuje decyzji. Po powrocie wystarczającej liczby menedżerów klaster wraca sam. Gdy nie wrócą, zostaje `docker swarm init --force-new-cluster`, czyli karta z Fiszki.

Menedżerowie potrzebują stałych adresów, bo adres ogłaszany (`--advertise-addr`) jest zapisany w klastrze. Dokumentacja ostrzega też, że zejście do jednego menedżera jest operacją niebezpieczną i niezalecaną [4].

Sieci overlay i porty

> **Pojęcie: sieć overlay.** Wirtualna sieć rozciągnięta nad kilkoma maszynami: kontenery z różnych węzłów widzą się w niej tak, jakby stały na jednym przełączniku. Ruch jest pakowany w tunel VXLAN (UDP 4789) i przechodzi przez zwykłą sieć serwerową.

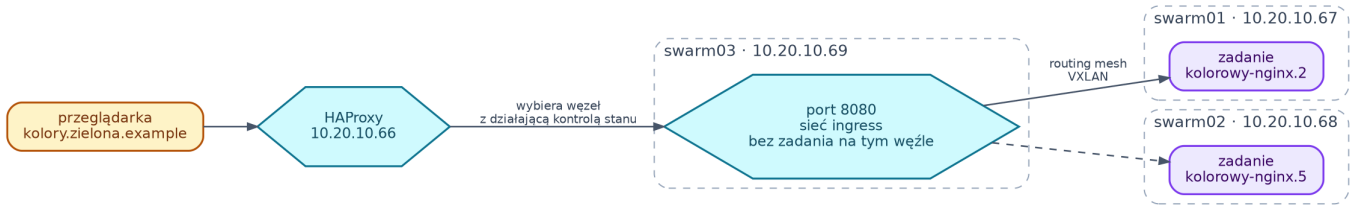
Swarm potrzebuje trzech portów między węzłami: 2377/tcp dla komunikacji menedżerów, 7946 tcp i udp dla wykrywania węzłów i 4789/udp dla ruchu overlay [6]. Dokumentacja ostrzega, że VXLAN nie ma uwierzytelniania, więc port 4789 otwieramy tylko dla zaufanej sieci i nigdy na zaporze brzegowej [6]. Dla sieci szyfrowanych (`--opt encrypted`) przepuszczamy też protokół IP 50 (ESP) [6].

W laboratorium otwieramy te porty wyłącznie dla trzech adresów węzłów, w osobnej strefie firewalld:

```
sudo firewall-cmd --permanent --new-zone=swarm
sudo firewall-cmd --permanent --zone=swarm --add-source=10.20.10.67/32
sudo firewall-cmd --permanent --zone=swarm --add-source=10.20.10.68/32
sudo firewall-cmd --permanent --zone=swarm --add-source=10.20.10.69/32
sudo firewall-cmd --permanent --zone=swarm --add-port=2377/tcp --add-port=7946/tcp --add-port=7946/udp --add-port=4789/udp
sudo firewall-cmd --reload
```

Routing mesh

> **Pojęcie: routing mesh.** Mechanizm, w którym opublikowany port usługi jest otwarty na każdym węźle klastra, także na tym, na którym nie ma żadnego zadania tej usługi. Węzeł przekazuje połączenie siecią ingress do któregoś z zadań [7].



Rysunek 2. Routing mesh: HAProxy wybiera dowolny węzeł z działającą kontrolą stanu, a ten przekazuje połączenie siecią ingress do zadania na innym węźle. (kliknięcie otwiera pełny rozmiar)

Dla nas ma to dwie konsekwencje. Po pierwsze, HAProxy może wskazywać wszystkie trzy węzły i nie musi wiedzieć, gdzie stoją zadania: kontrola portu 8080 wystarcza. Po drugie, port 8080 jest zajęty na wszystkich trzech maszynach, więc inny proces (na przykład Traefik) nie może go użyć. Dokumentacja pokazuje też tryb `mode=host`, w którym port jest otwarty tylko tam, gdzie stoi zadanie, bez routing mesh [7].

Warto też wiedzieć, że klient łączący się przez routing mesh widzi w aplikacji adres węzła, a nie swój własny. Gdy aplikacja potrzebuje adresu klienta, publikujemy port w trybie `host` albo stawiamy przed nią proxy z nagłówkami.

Sekrety i konfiguracje

> **Pojęcie: sekret Swarm.** Dane poufne (hasło, klucz), które trafiają do klastra raz, są szyfrowane w logu Raft i przesyłane szyfrowanym kanałem, a kontener widzi je jako plik w pamięci pod `/run/secrets/<nazwa>` [8]. Rozmiar jednego sekretu to najwyżej 500 KB, a sekretu używanego przez działającą usługę nie da się usunąć [8].

> **Pojęcie: konfiguracja Swarm (config).** To samo co sekret, tylko bez szyfrowania w spoczynku: plik niepoufny, na przykład skrypt albo fragment konfiguracji, zapisany w Raft i podmontowany w kontenerze pod wskazaną ścieżką, domyślnie z prawami `0444` [9]. Konfiguracja jest niezmienialna: żeby ją zmienić, tworzymy nową pod inną nazwą i podmieniamy w usłudze [9].

Na stronie Hasła i sekrety poza repozytorium: .env, Docker secrets i Vaultwarden opisaliśmy „sekrety Compose", które są plikami z dysku hosta. Sekret Swarm to inny mechanizm z tą samą ścieżką `/run/secrets`: leży w klastrze, nie na dysku. Fragment stosu, który z niego korzysta:

```
services:
  aplikacja:
    image: NazwaObrazu:Wersja
    secrets:
      - NazwaSekretu
    environment:
      TOKEN_FILE: /run/secrets/NazwaSekretu

secrets:
  NazwaSekretu:
    external: true
```

Wartość `external: true` mówi, że sekret już istnieje (karta z Fiszki), a stos go tylko używa. Dzięki temu hasło nie leży w żadnym pliku w Gitea.

Stos zamiast Compose: co się zmienia

Polecenie `docker stack deploy` czyta plik w formacie zbliżonym do Compose, ale według dokumentacji używa starszego formatu wersji 3 i część kluczy ignoruje [10]. W obrazie z

poprzedniej strony zapisaliśmy w naszej konwencji kilka rzeczy, które tu przestają działać.

Klucz z pliku Compose	Co robi `docker stack deploy`	Co robimy
`build`	ostrzeżenie `Ignoring unsupported options: build`, obrazu nie zbuduje	budujemy obraz osobno i wysyłamy do rejestru, bo każdy węzeł musi go pobrać [10]
`container_name`	ostrzeżenie, że ustawianie nazwy kontenera nie jest wspierane	nazwa zadania to `stos_usluga.numer.id`, nie wpisujemy własnej
`restart`	ostrzeżenie `Ignoring unsupported options: restart`	`deploy.restart_policy`
`depends_on`	dokumentacja mówi, że kolejność nie jest wymuszana [10] (w naszej próbie bez ostrzeżenia)	aplikacja sama ponawia połączenia, albo `healthcheck`
`volumes` z bind mount	katalog źródłowy musi istnieć na każdym węźle, który dostanie zadanie	patrz sekcja o danych
`ports`	dostępne krótkie i długie zapisy, domyślnie `mode: ingress`	długi zapis z `mode: host`, gdy nie chcemy routing mesh

Ostrzeżenia z dwóch pierwszych wierszy zaobserwowaliśmy w Docker CLI 29.4.3 w piaskownicy: `Ignoring unsupported options: build, restart` oraz osobny komunikat o `container_name`. Plik stosu sprawdzamy bez wdrażania poleceniem `docker stack config -c plik.yaml`, które wypisuje plik po podstawieniu zmiennych i zgłasza błędy składni.

Konwencja Zielonej opiera się na `container_name` i bind mountach pod `/docker/apps/<nazwa>`. Na stosie żadne z nich nie działa tak samo, więc przeniesienie istniejących plików Compose z `docker01` oznacza przepisanie ich, nie skopiowanie. Dlatego nie przenosimy tak całej półki aplikacji.

Dane: dlaczego to jest problem

Swarm potrafi odtworzyć kontener na innym węźle, ale nie potrafi przenieść jego danych. Wolumen lokalny (`local`) i bind mount leżą na jednej maszynie, a zadanie, które po awarii wstanie gdzie indziej, zobaczy pusty katalog albo błąd.

Sposób	Co daje	Cena
Przypięcie do węzła (`placement.constraints` z etykietą węzła)	dane zostają tam, gdzie są	awaria tego węzła zatrzymuje usługę, więc to nie jest wysoka dostępność

Sposób	Co daje	Cena
Wolumen NFS (sterownik `local` z opcjami NFS)	wszystkie węzły widzą te same pliki	serwer NFS jest nowym punktem awarii, a SQLite i bazy na NFS są ryzykowne
Cluster volumes (CSI)	wolumen, który Swarm sam przydziela węzłom przez wtyczkę CSI	funkcja nowa: dokumentacja Moby mówi, że nie obsługuje migawek, klonowania ani rozszerzania, a wtyczka musi stać na wszystkich menedżerach; dojrzałość w 2026 roku do potwierdzenia [11]
Replikacja w aplikacji	baza ma własne repliki (`pg01`, Mongo, Cassandra)	działa niezależnie od orkiestratora i jest to właściwa droga

Etykieta węzła pod przypięcie nadaje się w tej komendzie:

```
docker node update --label-add dane=tak swarm01
```

A w pliku stosu do usługi dopisujemy:

```
    deploy:
      placement:
        constraints:
          - node.labels.dane == tak
```

Wniosek jest ten sam, co na stronie o replikacji danych: Swarm nie jest odpowiedzią na pytanie „jak mieć dane na drugim hoście”. Dla aplikacji bezstanowych (kolorowy nginx, frontend bez sesji, proste API) jest dobrym narzędziem, a dla aplikacji z danymi w plikach nie.

Aktualizacje kroczące

Sekcja `update_config` w pliku stosu opisuje, jak Swarm wymienia zadania po zmianie obrazu albo konfiguracji. Nasze ustawienia: dwa zadania naraz (`parallelism: 2`), 10 sekund przerwy (`delay: 10s`), nowe zadanie startuje przed zatrzymaniem starego (`order: start-first`), a nieudana aktualizacja wycofuje się sama (`failure_action: rollback`).

Parametr	Znaczenie	Wartość w naszym stosie
`parallelism`	ile zadań wymieniamy jednocześnie	2
`delay`	przerwa między grupami	10 sekund
`order`	`stop-first` (domyślnie) albo `start-first`	`start-first`, bo nginx może chwilę stać dwa razy

Parametr	Znaczenie	Wartość w naszym stosie
`failure_action`	`pause`, `continue` albo `rollback`	`rollback`
`monitor`	jak długo obserwujemy nowe zadanie po starcie	15 sekund

Wersje `start-first` i kontrola stanu (`healthcheck` w pliku) działają razem: nowe zadanie liczy się jako dobre dopiero, gdy przejdzie kontrolę przez czas `monitor`. Bez `healthcheck` Swarm uznaje zadanie za zdrowe zaraz po starcie procesu, więc aplikacja, która startuje powoli, dostaje ruch za wcześnie.

Konfiguracje są niezmienialne, więc zmiana skryptu kolorów to nowa konfiguracja pod nową nazwą (na przykład `kolor_sh_v2`) i `docker stack deploy`, które wymienia zadania kroczą. Ta sama zasada dotyczy sekretów [9].

Przebieg

> W naszym środowisku dane aplikacji trzymamy w `/docker/apps/<nazwa>`, a plik Compose w `/docker/compose/<nazwa>/compose.yaml`. To konwencja, nie wymóg Dockera: można ją zmienić, byle w całej firmie była jedna. Katalog `/docker` to osobny wolumen `vgDocker/lvDocker`, żeby pełny dysk aplikacji nie zatrzymał systemu.

Zanim zaczniemy

Laboratorium wymaga trzech maszyn wirtualnych z Rocky Linux 9 i tą samą wersją Dockera co `docker01` (strona Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker): `swarm01` (`10.20.10.67`), `swarm02` (`10.20.10.68`), `swarm03` (`10.20.10.69`). Dwie maszyny stoją na jednym z hostów Hyper-V, więc awaria tego hosta zabiera kworum Raft do czasu restartu maszyn przez klaster Hyper-V. Tę samą pułapkę rozbieramy dokładnie na stronie k3s z trzema węzłami serwerowymi: etcd, adres wirtualny API i Longhorn, a w laboratorium przyjmujemy ją świadomie.

Adresy `10.20.10.67` do `10.20.10.69` są w puli przydzielonej w tym przebiegu i sprawdziliśmy w całej bazie, że nie są nigdzie użyte. Te same trzy adresy używa laboratorium k3s (jako `k3s02`, `k3s03` i adres wirtualny API), więc oba laboratoria stawiamy po kolei, nie równocześnie. Przed laboratorium k3s maszyny Swarm kasujemy (`docker swarm leave --force` i usunięcie maszyn) albo instalujemy od nowa.

Przed rozpoczęciem potrzebujemy też pary `lb01` i `lb02` ze strony Równoważenie i adres wirtualny: keepalived i HAProxy na parze lb01 i lb02. Bez niej testujemy `curl` wprost na adresy węzłów.

1. Strefa zapory na każdym węźle

Wykonujemy komendy z sekcji „Sieci overlay i porty” na wszystkich trzech maszynach. Port 8080/tcp (nasz opublikowany port nginksa) otwieramy tylko dla `lb01` i `lb02` oraz dla stacji administratora:

```
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source
address="10.20.10.64/31" port port="8080" protocol="tcp" accept'
sudo firewall-cmd --reload
```

Docker publikuje porty własnymi regułami, więc po wdrożeniu sprawdzamy, czy 8080 faktycznie nie odpowiada z hosta spoza listy (patrz uwaga o `DOCKER-USER` na stronie o HAProxy).

2. Założenie klastra

Na `swarm01` wykonujemy kartę „docker swarm init” z Fiszki. Na tej samej maszynie odczytujemy token menedżera, a na `swarm02` i `swarm03` wykonujemy `docker swarm join` z tym tokenem. Na koniec sprawdzamy listę węzłów:

```
docker node ls
```

Mają być trzy wiersze `Ready` i `Active`, z jednym `Leader` i dwoma `Reachable`.

> Polecenie `docker swarm init --autolock` dodaje blokadę: po restarcie menedżera trzeba odblokować go kluczem, a bez klucza jego dysk nie ujawnia kluczy szyfrujących Raft. W laboratorium go nie włączamy, a przy produkcyjnym klastrze zdecydujemy po sprawdzeniu opisu w dokumentacji.

3. Plik stosu z kolorowym nginksem

Skrypt wyboru koloru, ten sam co w k3s, ale z nazwą zadania Swarm zamiast nazwy poda. Zapisujemy go jako plik `40-kolor.sh` obok pliku stosu:

```
sudo mkdir -p /docker/compose/kolory-swarm
sudo vim /docker/compose/kolory-swarm/40-kolor.sh
```

```
#!/bin/sh
# 40-kolor.sh: wybiera kolor po nazwie zadania Swarm i zapisuje stronę dla nginx.
# Skrypt uruchamia sam obraz nginx przy starcie kontenera (katalog /docker-entrypoint.d).
set -e
KOLORY="czerwony:#ef4444 zielony:#22c55e niebieski:#3b82f6 zolty:#eab308 fioletowy:#a855f7
pomaranczowy:#f97316"
LICZBA=$(echo $KOLORY | wc -w)
HASH=$(echo -n "$TASK_NAME" | cksum | cut -d' ' -f1)
INDEKS=$((HASH % LICZBA))
WYBRANY=$(echo $KOLORY | cut -d' ' -f$((INDEKS + 1)))
NAZWA=$(echo $WYBRANY | cut -d':' -f1)
HEX=$(echo $WYBRANY | cut -d':' -f2)
STRONA=/usr/share/nginx/html/index.html
echo "<html><body style='background-color:$HEX; font-family:sans-serif; text-align:center;
padding-top:15%;'>" > $STRONA
echo "<h1>Zadanie: $TASK_NAME</h1>" >> $STRONA
echo "<h2>Kolor: $NAZWA</h2>" >> $STRONA
echo "<h3>Wezel: $NODE_NAME</h3>" >> $STRONA
echo "</body></html>" >> $STRONA
```

Plik stosu:

```
sudo vim /docker/compose/kolory-swarm/kolory.yaml
```

```
services:
  kolorowy-nginx:
    image: nginx:1.30.5
    environment:
      - TASK_NAME={{.Task.Name}}
      - NODE_NAME={{.Node.Hostname}}
    configs:
      - source: kolor_sh
        target: /docker-entrypoint.d/40-kolor.sh
        mode: 0755
    ports:
      - target: 80
        published: 8080
        protocol: tcp
        mode: ingress
    healthcheck:
```

```
      test: ["CMD-SHELL", "bash -c 'echo > /dev/tcp/127.0.0.1/80'"]
      interval: 10s
      timeout: 3s
      retries: 3
      start_period: 5s
    deploy:
      replicas: 6
      update_config:
        parallelism: 2
        delay: 10s
        order: start-first
        failure_action: rollback
        monitor: 15s
      rollback_config:
        parallelism: 2
        order: start-first
      restart_policy:
        condition: on-failure
        delay: 5s
      resources:
        limits:
          cpus: "0.20"
          memory: 64M
        reservations:
          cpus: "0.05"
          memory: 32M
      placement:
        max_replicas_per_node: 3

configs:
  kolor_sh:
    file: ./40-kolor.sh
```

Różnice względem strony o k3s są cztery. Zamiast kontenera inicjalizującego skrypt trafia do obrazu jako konfiguracja Swarm w katalogu `/docker-entrypoint.d`, który oficjalny obraz nginx uruchamia przy starcie, jeśli plik jest wykonywalny (stąd `mode: 0755`). Nazwę zadania i węzła dostają zmienne środowiskowe z szablonów `{{.Task.Name}}` i `{{.Node.Hostname}}`, które Swarm podstawia przy tworzeniu zadania. Zamiast sond HTTP jest jeden `healthcheck`, który sprawdza, czy nginx przyjmuje połączenia (`bash` jest w obrazie opartym o Debiana, ale nie w wariancie `alpine`); liczba kopii to sześć, z limitem trzech na węzeł, żeby po wyłączeniu jednego węzła pozostałe dwa przyjęły wszystkie.

> Jeśli `healthcheck` używa `bash`, nie zamieniamy obrazu na `nginx:1.30.5-alpine` bez zmiany sprawdzenia. W Alpine nie ma `bash`, kontrola kończy się błędem, a aktualizacja z `failure_action: rollback` zostanie wycofana, co jest dobrym ćwiczeniem (patrz niżej).

```
docker stack config -c /docker/compose/kolory-swarm/kolory.yaml > /dev/null && echo "plik
stosu jest poprawny"
```

4. Wdrożenie i sprawdzenie

```
cd /docker/compose/kolory-swarm
docker stack deploy -c kolory.yaml kolory
docker stack services kolory
docker service ps kolory_kolorowy-nginx --format '{{.Name}} {{.Node}} {{.CurrentState}}'
```

Przykład po kilkunastu sekundach:

```
ID             NAME                    MODE         REPLICAS               IMAGE
PORTS
x7y8z9a0b1c2   kolory_kolorowy-nginx   replicated   6/6 (max 3 per node)   nginx:1.30.5
*:8080->80/tcp
```

Liczba `6/6` znaczy, że wszystkie zadania działają. Pierwsze pobranie obrazu na każdym węźle trwa chwilę, bo każdy pobiera go sam z rejestru.

5. HAProxy przed klastrem

Do pliku `haproxy.cfg` z poprzedniej strony dopisujemy backend i regułę w `fe_http` (pod linią `use_backend be_mailcow_http`):

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

```
backend be_swarm_kolory
    mode http
    balance roundrobin
    option httpchk
    http-check send meth GET uri / ver HTTP/1.1 hdr Host kolory.zielona.example
    http-check expect status 200
    default-server inter 3s fall 3 rise 2
```

```
    server swarm01 10.20.10.67:8080 check
    server swarm02 10.20.10.68:8080 check
    server swarm03 10.20.10.69:8080 check
```

Po `haproxy -c` i `reload` na obu maszynach `lb` zakładamy rekord `kolory` jako `CNAME` do `lb-ha.ad.zielona.example` (komenda z strony o HAProxy). Ten sam plik z połączonym blokiem sprawdziliśmy poleceniem `haproxy -c` w wersji 2.8.16.

6. Weryfikacja

```
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
```

Dwanaście osobnych połączeń daje wymieszane kolory. Przeglądarka z `keep-alive` pokazuje jeden kolor do czasu otwarcia nowej karty, tak jak opisaliśmy to dla k3s. Dla porównania nazwy zadań z listy `docker service ps` wchodzą do skrótu, więc po skalowaniu (`docker service scale kolory_kolorowy-nginx=6` jeszcze raz) pojawiają się nowe kolory.

7. Aktualizacja krocząca i wycofanie

Zmieniamy skrypt kolorów (dopisujemy kolor do listy `KOLORY`), zapisujemy go i nadajemy konfiguracji nową nazwę w pliku stosu:

```
configs:
  kolor_sh:
    name: kolor_sh_v2
    file: ./40-kolor.sh
```

```
docker stack deploy -c kolory.yaml kolory
watch -n 2 "docker service ps kolory_kolorowy-nginx --format '{{.Name}} {{.CurrentState}}' |
head -n 14"
```

Swarm wymienia po dwa zadania naraz, a nowe startuje przed zatrzymaniem starego. W pętli `curl` z kroku 6 nie powinno być przerw. Cofnięcie jednym poleceniem to karta „docker service rollback” z Fiszki.

Ćwiczenie z awarią aktualizacji: zmieniamy w pliku obraz na `nginx:1.30.5-alpine` i wdrażamy. Oczekujemy, że `healthcheck` z `bash` zawiedzie, aktualizacja się zatrzyma i dzięki `failure_action:`

`rollback` wróci do poprzedniej wersji. To założenie wynika z dokumentacji, a nie z próby na trzech węzłach, więc sprawdzamy je w laboratorium i opisujemy faktyczny przebieg w tabeli wyników.

Test awarii

Wszystkie testy robimy z pętlą `curl` z kroku 6 w drugim oknie i zapisujemy czas przerwy.

1. Planowe wyłączenie węzła. `docker node update --availability drain swarm03`, potem sprawdzamy listę zadań: pracują tylko `swarm01` i `swarm02`, po trzy zadania. Powrót: `--availability active` i `docker service update --force kolory_kolorowy-nginx`, żeby zadania rozłożyły się znów równomiernie.
2. Awaria węzła. Na hoście, który jest właścicielem roli (`Get-ClusterGroup -Name swarm02` pokazuje go w kolumnie `OwnerNode`), wykonujemy `Stop-VM -Name swarm02 -ComputerName NazwaHosta -TurnOff -Force`. `Stop-VM -TurnOff` odcina maszynie zasilanie bez zamykania systemu, czyli udaje awarię sprzętu, a `Stop-ClusterGroup` wyłącza rolę według akcji wyłączenia ustawionej w klastrze, która może zapisać stan maszyny zamiast ją wyłączyć (do potwierdzenia w ustawieniach roli), więc nie jest twardą awarią. HAProxy oznacza `swarm02` jako `DOWN` po kilku kontrolach, a Swarm uznaje węzeł za `Down` i tworzy brakujące zadania na pozostałych. Czas do uznania węzła za niedziałający zależy od okresu sprawdzania ustawianego opcją `docker swarm update --dispatcher-heartbeat`, więc mierzymy go w laboratorium.
3. Utrata kworum. Wyłączamy dwie z trzech maszyn. Na ocalałej `docker node ls` przestaje odpowiadać, a nie da się wdrożyć niczego nowego, ale istniejące zadania dalej pracują [4]. Kolejnych maszyn nie przywracamy od razu: sprawdzamy, czy strona nadal odpowiada przez `lb`, a potem włączamy maszyny i patrzymy, jak klaster wraca sam.
4. Wyzwanie ostatniej szansy (tylko w laboratorium). Z jednym menedżerem i bez szans na powrót pozostałych wykonujemy kartę „docker swarm init --force-new-cluster” i dołączamy węzły od nowa.

Utrzymanie tego klastra (stan Raft, autolock, kopia `/var/lib/docker/swarm`, odzyskanie kworum, rotacja certyfikatów i aktualizacja silnika) opisuje strona Docker Swarm w wysokiej dostępności: Raft, autolock, kopia i odzyskanie kworum, a pomiar awarii równolegle z k3s i kubeadm strona Ćwiczenie awarii na trzech klastrach: Swarm, k3s i kubeadm równolegle.

Co sprawdziliśmy w piaskownicy

W piaskownicy nie mieliśmy trzech maszyn, więc nie sprawdziliśmy kworum, routing mesh między węzłami ani rozkładu zadań. Uruchomiliśmy Docker Engine 29.4.3 z jednym węzłem Swarm i sprawdziliśmy mniejsze rzeczy.

Co	Wynik

`docker stack config` na pliku stosu z tej strony	poprawny, zmienne `{{...}}` zachowane
`docker stack deploy` z `container_name`, `restart`, `depends_on`, `build`	ostrzeżenia o `build`, `restart` i `container_name`, brak ostrzeżenia o `depends_on`
Wdrożenie z szablonami `{{.Task.Name}}`, `{{.Node.Hostname}}`, `{{.Task.Slot}}`	zmienne środowiskowe podstawione w zadaniach
Konfiguracja z `mode: 0755` pod `/docker-entrypoint.d/`	plik wykonywalny w kontenerze (`-rwxr-xr-x`), skrypt uruchamia się
Skrypt `40-kolor.sh` z udawanymi nazwami zadań	składnia poprawna (`sh -n`, `bash -n`), różne nazwy dają różne kolory
`docker service inspect --pretty`	widoczne `UpdateConfig` (start-first, rollback) i `max replicas per node`

Obrazu `nginx` nie mogliśmy pobrać z Docker Hub w tej sesji, więc zadania uruchamialiśmy na lokalnym obrazie zastępczym. Sieć ingress w piaskownicy zgłaszała błąd o nakładających się pulach adresów, więc zadania ze stosem z opublikowanym portem nie wystartowały. To ograniczenie piaskownicy, nie pliku, ale właśnie dlatego pełny test na trzech maszynach pozostaje do zrobienia.

Kiedy Swarm ma sens dla Zielonej

Pytanie	Swarm	Dwa hosty z keepalived (strona o przenoszeniu)	k3s z trzema węzłami
Minimalna liczba maszyn	trzy menedżerowie	dwa hosty	trzy serwery
Konwencja `/docker/apps` i `container_name`	do przepisania	działa bez zmian	do przepisania na manifesty
Dane na drugim hoście	nie rozwiązuje	rsync albo ZFS	Longhorn
Aktualizacje kroczące	tak, w pliku stosu	nie, to rola aplikacji	tak, Deployment
Samonaprawa po awarii zadania	tak	skrypty	tak
Ekosystem narzędzi	niewielki	zwykły Docker	duży
Krzywa nauki dla dwóch administratorów	mała	najmniejsza	największa

Swarm ma sens, gdy aplikacje są bezstanowe lub trzymają dane w zewnętrznej bazie, gdy mamy co najmniej trzy maszyny na menedżerów i gdy zespół zna już Dockera, a Kubernetesa nie chce się uczyć. W Zielonej takich aplikacji jest dziś niewiele: Gitea, Vaultwarden i MailCow trzymają dane w plikach, a NetBox i Graylog zależą od baz w kontenerach.

Swarm nie ma sensu, gdy głównym problemem są dane, a nie kopie aplikacji, gdy mamy tylko dwa hosty i gdy przepisanie plików Compose zjadłoby więcej czasu, niż daje zysk. W Zielonej to

przypadek `docker01` i `docker02`, więc Swarm nie jest naszym pierwszym wyborem. Zostaje laboratorium i ewentualnie przyszłe aplikacje zespołu Oli, pisane od początku bezstanowo. Jeśli ma powstać klaster, który przeżyje awarię węzła, bliższy naszym potrzebom jest k3s opisany na następnej stronie, bo ma przyzwoitą odpowiedź na dane (Longhorn) i dużo większy ekosystem.

Wycofanie zmian

```
docker stack rm kolory
docker config rm kolor_sh_v2
```

Na `swarm02` i `swarm03` wykonujemy `docker swarm leave`, a na `swarm01` (ostatnim menedżerze) `docker swarm leave --force`. Z `haproxy.cfg` na `lb01` i `lb02` usuwamy backend `be_swarm_kolory` i regułę w `fe_http`, po czym `haproxy -c` i `reload`. Rekord `kolory` w DNS zostaje do laboratorium k3s. Strefę firewalld `swarm` usuwa `sudo firewall-cmd --permanent --delete-zone=swarm` i `--reload`.

Częste błędy

Objaw	Przyczyna	Co robimy
`docker swarm join` kończy się błędem połączenia	zapora zatrzymuje 2377/tcp albo adres menedżera jest zły	strefa `swarm` na wszystkich węzłach, `curl -v telnet://10.20.10.67:2377`
Zadania na różnych węzłach nie widzą się w sieci overlay	zablokowane 4789/udp albo 7946	otwieramy porty z sekcji o sieci overlay, na każdym węźle
Zadanie ma stan `Rejected` albo `no suitable node`	limit `max_replicas_per_node`, brak zasobów albo węzeł w `drain`	`docker service ps ... --no-trunc`, `docker node ls`
Stos kończy się błędem o niestniejącym katalogu	bind mount ze ścieżką, której nie ma na węźle	katalog tworzymy na każdym węźle albo przypinamy usługę do węzła
Po powrocie węzła z `drain` zadania nie wracają	Swarm nie rozkłada zadań z powrotem sam	`docker service update --force NazwaUslugi`
`docker service update --image` wisi	`healthcheck` nie przechodzi albo za krótki `start_period`	`docker service ps`, logi zadania, `docker service rollback`
`docker node ls` kończy się błędem po wyłączeniu dwóch maszyn	utrata kworum (mniej niż 2 z 3)	włączamy brakujące maszyny, w ostateczności `--force-new-cluster`
Aplikacja nie widzi adresu klienta	routing mesh podmienia adres źródłowy	port w trybie `host` albo proxy z nagłówkami
Hasło trafiło do historii powłoki	sekret tworzony komendą z hasłem w argumencie	`read -s` i potok, jak w karcie z Fiszki

□□ Ćwiczenia

1. Plik Compose z `docker01` ma `container_name: gitea`, `restart: always` i bind mount `/docker/apps/gitea/data`. Wypisujemy, co z tego zadziała na stosie Swarm i co trzeba zmienić.

Odpowiedź

`container_name` i `restart` są ignorowane (z ostrzeżeniami), więc zamiast `restart` piszemy `deploy.restart_policy`. Bind mount zadziała tylko na węźle, na którym katalog istnieje, więc zadanie na innym węźle zobaczy pusty katalog albo błąd. Dla Gitea z danymi w plikach należy zachować aktywny-pasywny z keepalived (strona o przenoszeniu kontenerów), a nie przenosić jej na Swarm.

1. Dlaczego dwóch menedżerów to gorszy układ niż jeden?

Odpowiedź

Większość z dwóch głosów to dwa głosy, więc utrata któregokolwiek menedżera zatrzymuje klaster. Przy jednym menedżerze awaria też go zatrzymuje, ale mamy jedną maszynę do pilnowania zamiast dwóch, z których każda może przerwać pracę. Trzy maszyny tolerują jedną awarię.

1. Po `docker node update --availability drain swarm03` wszystko działa. Dlaczego po powrocie (`active`) na `swarm03` nie ma żadnych zadań i co robimy?

Odpowiedź

Swarm przenosi zadania tylko wtedy, gdy musi, więc po powrocie węzła nie przesuwa istniejących. Wykonujemy `docker service update --force kolory_kolorowy-nginx`, co wymienia zadania i rozkłada je ponownie według `max_replicas_per_node`.

1. Konfiguracja Swarm jest niezmienialna. Jak zmienić skrypt `40-kolor.sh` w działającej usłudze bez przerwy?

Odpowiedź

Tworzymy nową konfigurację pod nową nazwą (`kolor_sh_v2` w polu `name` pliku stosu) i wdrażamy stos. Swarm wymienia zadania według `update_config`, a stara konfiguracja zostaje w

klastrze, dopóki jej nie usuniemy. Wycofanie to powrót do poprzedniej nazwy.

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 · k3s z trzema węzłami serwerowymi: etcd, adres wirtualny API i Longhorn · k3s w laboratorium: pierwszy deploy, nginx w kolorach · Hasła i sekrety poza repozytorium: .env, Docker secrets i Vaultwarden · Instalacja Dockera na Rocky Linux 9 z osobnym wolumenem /docker · Docker Swarm w wysokiej dostępności: Raft, autolock, kopia i odzyskanie kworum

Źródła:

1. mirantis.com: Mirantis guarantees long-term support for Swarm, 1 lipca 2025
2. docs.docker.com: Docker Engine 29 release notes (wydanie 29.8.2 z 30 września 2026)
3. virtualizationhowto.com: Is Docker Swarm still safe in 2026, marzec 2026 (blog, nie dokument projektu)
4. docs.docker.com: Administer and maintain a swarm of Docker Engines
5. docs.docker.com: How nodes work
6. docs.docker.com: Getting started with Swarm mode
7. docs.docker.com: Use swarm mode routing mesh
8. docs.docker.com: Manage sensitive data with Docker secrets
9. docs.docker.com: Store configuration data using Docker Configs
10. docs.docker.com: Deploy a stack to a swarm
11. github.com/moby/moby: docs/cluster_volumes.md
12. github.com: docker/swarm (Swarm Classic) (początek w 2014 roku, archiwum od 1 lutego 2021, tryb Swarm od Dockera 1.12 na bazie SwarmKit; odczyt 18 października 2026)
13. versionlog.com: Docker Engine 1.12 (wydanie 1.12.0 z 28 lipca 2016, tryb Swarm w silniku)

Revision #15
Created 2026-10-03 13:44:09 UTC by Tomasz Konieczny
Updated 2026-10-26 14:21:51 UTC by Tomasz Konieczny