JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 23 (2022) 97–104
Received: 7 March 2022
Accepted: 10 April 2022

Comparison of virtualization methods at operating system level

Porównanie metod wirtualizacji na poziomie systemu operacyjnego

Łukasz Grula*, Paweł Powroźnik

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The aim of the work is comparative analysis of three tools for application container's orchestration: Kubernetes 1.2.2, Docker Swarm 1.24 and Nomad Hashicorp 1.2.0. For this purpose, test application was implemented, responding requests, then it was contenerized using Docker. For each tool, the scenario aimed at measuring pods startup time. The research was repeated three times. During each repetition number of replics were increased. Simultaneously with startup time test, CPU load and memory strain were measured. In comparison also time of regeneration was taken into consideration, what was realized by gauging time of response for GET request. The analysis showed that Docker Swarm in terms of most of the criteria examined in this work turned out as the best orchestration tool.

Keywords: virtualization; Kubernetes; Docker; Nomad

Streszczenie

Przedmiotem tej pracy jest analiza porównawcza trzech narzędzi do orkiestracji kontenerów aplikacyjnych: Kubernetes 1.2.2, Docker Swarm 1.24 oraz Nomad Hashicorp 1.2.0. Zaimplementowano w tym celu aplikację, odpowiadającą na żądania, następnie skonteryzowano ją używając technologii Docker. Dla każdego z narzędzi powtórzono trzykrotnie scenariusz, który na celu miał zmierzenie czasu startu podów. Równocześnie z badaniem czasu startu przeprowadzono badanie dotyczące obciążenia podzespołów. W porównaniach uwzględniono też czas regeneracji repliki. Ostatnim doświadczeniem było zbadanie mechanizmów równoważenia obciążenia. Z przeprowadzonych analiz wynika, że Docker Swarm pod względem dużej części kryteriów rozpatrywanych w tej pracy okazał się najlepszym narzędziem orkiestracyjnym.

Słowa kluczowe: wirtualizacja; Kubernetes; Docker; Nomad

*Corresponding author

Email address: **lukasz.grula@pollub.edu.pl** (Ł. Grula)

1. Wstęp

W dzisiejszych czasach przy tworzeniu systemów coraz częściej zauważalny jest udział architektury mikroserwisowej. W związku z czym ważnymi aspektami przy zarządzaniu takimi systemami są: łatwość konfiguracji, wysoka dostępność, reagowanie na awarię, wysoka przepustowość, możliwość szybkiego wdrażania systemów do środowisk produkcyjnych. Spowodowało to potrzebę zmiany podejścia stosowanego do tej pory - wirtualizacji, która wiązała się z dużą liczbą konfiguracji oraz małą uniwersalnością w odtwarzaniu środowiska. W dodatku wirtualizacja wymuszała większe wykorzystanie procesora, pamięci operacyjnej, a czas uruchamiania był na tyle duży, aby zapewnienie wysokiej dostępności i ciągłości funkcjonowania systemu było kłopotliwe.

Kolejnym rozwiązaniem wpływającym na usprawnienie funkcjonowania systemów było zastosowanie metod konteneryzacji, a więc odejście od używania maszyn wirtualnych i zastosowanie wirtualizacji systemów operacyjnych dzięki kontenerom. Obrazy kontenerów funkcjonują przy jądrze tego samego systemu operacyjnego w odróżnieniu od wirtualizacji.

Jednak rozwiązanie to nie oferowało wysokiej dostępności, reakcji na awarię czy automatycznego równoważenia obciążenia. Wtedy wprowadzone zostało pojęcie orkiestracji kontenerów, czyli zautomatyzowania procesów kontenerowych, a jednocześnie dostarczenie szeregu ułatwień w zapewnieniu ciągłej stabilności systemów i zarządzania nimi.

Obecnie na rynku pojawia się coraz więcej narzędzi, które wspomagają orkiestrację kontenerów, co pokazuje jak ważny jest to aspekt. Jednakże powoduje to również trudności w podjęciu decyzji, które narzędzie będzie najbardziej wydajne dla konkretnego systemu informatycznego. W związku z tym powstają artykuły oraz prace, które podejmują tą tematykę i próbują rozwiązać problemy związane z wyborem odpowiedniej technologii. W ramach przeprowadzonych badań przeanalizowano i porównano trzy narzędzia do orkiestracji kontenerów: Kubernetes (K8S), Docker Swarm oraz Nomad Hashicorp, które posiadają podobne funkcjonalności.

2. Przegląd literatury

W przypadku systemów bazujących na architekturze mikroserwisowej kluczowym jest odpowiednie ich wdrażanie i zarządzanie nimi. Wybrano i opisano więc artykuły, które skupiają się na tematyce wirtualizacji, konteneryzacji i orkiestracji.

Artykuł [1] porównuje zagadnienia wirtualizacji i konteneryzacji. Porównanie przedstawia między innymi zalety konteneryzacji w aspektach czasu startu, co autorzy przedstawiają jako jeden z głównych powodów wyparcia wirtualizacji przez kontenery. Kolejne zalety to między innymi możliwość dzielenia plików, bibliotek między kontenerami. Jest to cecha, która wpływa na znacznie mniejsze możliwości komunikacyjne między wirtualizowanymi systemami niż kontenerami. Wnioski, które wysnuto na podstawie artykułu pokazują, że konteneryzacja wyparła wirtualizację w działaniach chmurowych ze względu na swoją szybkość, większe możliwości i niezależność od systemu.

W artykule [2] zestawiono ze sobą technologię Docker, KVM, LXC oraz XenServer. Porównano między innymi czas startu dwudziestu obrazów. Wyniki badania pokazują, że kontenery Docker uruchomiły się 10 razy szybciej niż systemy uruchomione na KVM. Przedstawiono również wydajność układów wejścia-wyjścia, gdzie rezultaty pokazały, że wydajność układów kontenerów Docker jest porównywalna z rozwiązaniami natywnymi, natomiast KVM jest niemal dwukrotnie wolniejszy. Wnioski autorów pokazują, że konteneryzacja znacznie automatyzuje pracę, a w dodatku poprzez dodatkową warstwę jest o wiele szybsza niż wirtualizacja.

Praca [3] przedstawia porównanie metod orkiestracji kontenerów – takich jak Docker Swarm, Kubernetes oraz Openshift. Pokazano wady i zalety każdego z narzędzi. Jednym z ważnych aspektów Docker Swarm jest wykorzystanie wbudowanego w środowisko Docker CLI, a więc rezygnacja z konieczności instalacji dodatkowych technologii. Jako wadę i zaletę jednocześnie autorzy przedstawiają lekkość narzędzia, co powoduje mniejszą liczbę funkcjonalności w stosunku do Kubernetesa lub Openshifta. Kubernetes posiada bardzo rozbudowany system, dzięki komendom narzędzia kubectl. Ostatnim porównanym narzędziem jest Openshift, który jak mówią autorzy jest zbudowany na podstawach Kubernetesa, jednakże jego celem są aplikacje biznesowe, pod które został dostosowany. Dzięki czemu jest szybszy w instalacji oraz posiada łatwiejszy interfejs graficzny. Natomiast jest płatny, w przeciwieństwie do Kubernetesa czy Docker Swarma, które są projektami opensource, oraz oferuje mniejszą uniwersalność.

Podmiotem badań kolejnej pracy [4] było przedstawienie aspektu orkiestracji oraz porównanie narzędzi takich jak Kuberentes, Docker Swarm, Apache Mesos oraz Cattle. Pierwszy scenariusz realizował zmierzenie czasu utworzenia klastra, sprawdzając złożoność orkiestratorów. W tym porównaniu najlepiej wypadł Cattle, z uwagi na to, że jest on natywnym narzędziem do orkiestracji dla Ranchera. Największy czas tworzenia klastra uzyskał Kubernetes, ze względu na wiele możliwości, które oferuje. Kolejnym badaniem było sprawdzenie czasu startu kontenerów aplikacji takich jak Jenkins oraz Gitlab, które miały odpowiednio jedną, dwie i cztery repliki. Sprawdzono czasy, gdy obrazy były pobierane z lokalnego repozytorium oraz z rejestru Dockera. Wyniki były porównywalne, ale przy pojedynczym obrazie najlepiej zadanie zrealizował Kubernetes. Natomiast zwiększając liczbę replik, uwzględniając lokalne repozytorium, Kubernetes osiągnął najlepszy wynik, natomiast przy obrazie znajdującym się w zewnętrznym repozytorium wypadł najgorzej. Ostatnim scenariuszem było porównanie mechanizmów pracy w sytuacji awarii, zdecydowano się przetestować dwa przypadki – awarię kontenera oraz awarię hosta. W pierwszym przypadku podczas awarii kontenera najlepiej wypadł Kubernetes, natomiast gdy doszło do awarii węzła rezultaty Cattle, Docker Swarm oraz Apache Mesos były niemal identyczne, a Kubernetes potrzebował o wiele więcej czasu na przywrócenie działającego węzła. Autorzy we wnioskach stwierdzili, że obecnie Kubernetes jest najbardziej kompletnym narzędziem do orkiestracji, natomiast w niektórych przypadkach ma to przełożenie na wydajność.

Ostatni analizowany artykuł [5] przedstawia porównanie orkiestratorów kontenerów bazujących na chmurze obliczeniowej. Na początku pracy przedstawiono tematykę konteneryzacji oraz orkiestracji i opisano porównywane narzędzia, czyli Docker Swarm i Kubernetes. W kolejnej części nawiązano do wcześniej powstałych prac o tej tematyce i przedstawiono tematykę badań przeprowadzonych w ramach artykułu. W kolejnym rozdziale autorzy przedstawiają testowane narzędzia, opisują ich systemy, sposób działania oraz najważniejsze moduły i komponenty. Do przeprowadzenia badań użyto dwóch testów sprawności – Phoronix Test Suite oraz LiDAR (*Light Detection and Ranging*). Za pomocą pierwszego z nich przetestowano takie parametry jak wydajność dysku podczas dużego obciążenia, testy pamięci RAM – szybkość, przepustowość i użycie cache - kompresję plików za pomocą różnych algorytmów, kodowanie audio i wideo oraz wydajność bazy danych z użyciem SQLite. Drugi z testów odpowiadał za przetwarzanie danych pozyskanych w ramach LiDAR. Przetestowano to używając klastrów składających się z dziewięciu węzłów dla obu narzędzi. Wnioski z przeprowadzonych testów wskazały, że Docker Swarm ma niemal pomijalny narzut czasowy w stosunku do natywnego silnika Docker, natomiast w przypadku Kubernetesa wynosił on 8.3%. Autorzy wskazali, że Swarm potrzebuje o wiele mniej dodatkowych narzędzi, aby uformować klaster składający się z kontenerów swojego natywnego silnika. Zaletami Docker Swarm są jego lekkość, łatwość tworzenia i konfiguracji klastra oraz fakt, że jest to natywne rozwiązanie silnika Docker. Natomiast Kubernetes finalnie posiada o wiele więcej możliwości, dodatków, oferuje bardziej rozbudowany system i jego wsparcie, co powoduje, że już od wielu lat budowane są na nim systemy produkcyjne.

3. Porównywane narzędzia

Na przestrzeni czasu powstają kolejne narzędzia do orkiestracji kontenerów, które są mniej lub bardziej popularne. Zadaniem tych narzędzi jest m. in. ułatwienie i przyspieszenie konfiguracji środowisk, zapewnienie wysokiej dostępności, wydajności oraz bezpieczeństwa systemów.

W ramach pracy przeprowadzono analizę porównawczą trzech narzędzi: Kubernetes, Docker Swarm, Nomad Hashicorp. Wybrano te orkiestratory z uwagi na ich popularność, co potwierdza liczba artykułów opisujących te technologie.

3.1. Kubernetes

Kubernetes jest przenośną i rozszerzalną platformą oprogramowania open source, która służy do zarządzania zadaniami i aplikacjami uruchamianymi w kontenerach [6]. Najmniejszą jednostką Kubernetesa jest Pod, który może zawierać jeden lub więcej kontenerów wraz z zasobami, które są przez nie wykorzystywane. Możliwością replikacji podów zarządzają kontrolery replikacji. Kolejną wielkością są węzły. To właśnie one składają się z podów oraz narzędzi do zarządzania nimi, węzeł może być maszyną wirtualną lub fizyczną. Natomiast najwyższym poziomem abstrakcji w Kubernetesie jest klaster, jest zdefiniowany jako grupa maszyn na których działa K8s. Kubernetes bazuje na wzorcu architektonicznym nazywanym master/slave, w którym klaster menadżera odpowiada za zarządzanie węzłami (zwane minionami). Kubernetes zapewnia konteneryzację aplikacji oraz dba o jej skalowalność, równoważenie obciążenia, zdrowie czy też alokację zasobów sprzętowych.

3.2. Docker Swarm

Docker Swarm jest jednym z wewnętrznych narzędzi, które należy do orkiestracji. Jest też nazywany natywnym narzędziem Dockera do klastrowania i zarządzania aplikacjami [7]. Pomaga to w zarządzaniu większą liczbą kontenerów, skalowaniu ich czy reagowaniu na awarię. Tak jak Kubernetes bazuje na modelu master/slave, gdzie węzeł menadżera zarządza pracą jednostek – węzłów pracowniczych. Możliwe jest skonfigurowanie węzłów menadżerów tak aby nie uruchamiały one zadań, a odpowiadały jedynie za zarządzaniem pracownikami. Węzły pracownicze uruchamiają serwisy, na których pracują kontenery. W dodatku istnieje też jeden węzeł lidera, który odpowiada za decyzje zarządzające trybem swarm. Dużą zaletą Docker Swarm jest to, że jest wbudowany w bazowy silnik Docker, a więc wymaga mało konfiguracji. Swarm posiada też wbudowany mechanizm balansowania obciążenia.

3.3. Nomad Hashicorp

Nomad jest orkiestratorem, który ułatwia organizacjom wdrażanie oraz zarządzanie skonteneryzowanymi i nieskonteneryzowanymi aplikacjami. Umożliwia on programistom wdrażanie aplikacji za pomocą infrastruktury jako usługi [8]. Nomad jest zbudowany wokół całego systemu Hashicorp, który integruje się z takimi narzędziami jak Terraform, Consul czy Vault. Pomaga to m. in. we wdrażaniu aplikacji, automatycznym wykrywaniu usług czy zarządzaniu bezpieczeństwem. Klaster Nomada uruchamia w sobie tzw. agenta, który może działać w trybie serwera lub klienta. Tryb jest swego rodzaju mózgiem klastra. Zarządza on klientami czy pracami oraz alokuje zadania. Serwery replikują dane między sobą i dokonują wyboru lidera, aby zapewnić wysoką dostępność usług. Wcześniej wspomniane zadania są najmniejszą jednostką klastra. Są one uruchamiane przez wbudowane lub zewnętrzne sterowniki. Zadania mogą tworzyć ich grupy, które potem składają się w tak zwane prace i to one działają na agentach. Twórcy Nomada podkreślają, że wspierają nie tylko skonteneryzowane aplikacje, ale również ułatwiają zarządzanie starszymi systemami.

4. Metoda badań

4.1. Opis eksperymentu

Analizę porównawczą narzędzi do orkiestracji kontenerów aplikacji Kubernetes, Docker Swarm i Nomad Hashicorp przeprowadzono na podstawie czterech scenariuszy badawczych:

1. Zmierzenie czasu uruchomienia startowego skonteneryzowanej aplikacji.
2. Zbadanie obciążenia pamięci, procesora i dysku.
3. Zbadanie skalowalności, mierząc czas odpowiedzi aplikacji korzystając z mechanizmów równoważenia obciążenia.
4. Zmierzenie czasów autoregeneracji narzędzi.

Pierwszy scenariusz badawczy dotyczący czasu startu aplikacji polegał na zmierzeniu czasu między rozpoczęciem procedury tworzenia kontenerów przez narzędzie orkiestracji, a momentu, kiedy aplikacja wewnątrz kontenerów będzie gotowa do działania. Przeprowadzono takie testy dla jednej i dziesięciu replik.

Drugi scenariusz tak jak poprzedni składał się z trzech testów dla wyżej wymienionych liczb replik. Polegało to na sprawdzeniu obciążenia procesora, pamięci oraz dysku przez aplikacje. Każde z narzędzi udostępniało natywnie metryki do zmierzenia powyższych obciążeń. Natomiast w przypadku Docker Swarm dla dokładniejszych pomiarów doinstalowano narzędzie Swarmprom, które udostępniało dokładniejsze metryki.

Kolejny scenariusz dotyczył zbadania mechanizmów autoregeneracji, polegało to na zmierzeniu czasu, między zlikwidowaniem jednej z replik, a przywróceniem nowej z gotową do działania aplikacją.

Ostatni scenariusz przedstawiał testy niezawodności używając mechanizmów zrównoważenia obciążenia. Przeprowadzono je za pomocą biblioteki Gatling. Dokonano pomiarów czasów obsługi żądań. Po przeprowadzeniu testów, Gatling wygenerował raport, który zawierał wykresy i tabele przedstawiające

wyniki i statystyki opisujące żądania i odpowiedzi. Do uruchomienia aplikacji wewnątrz narzędzi użyto szablonów przeznaczonych dla każdego z orkiestratorów.

4.2. Środowisko testowe

W tabeli poniżej przedstawiono opis środowiska testowego, użytego do przeprowadzenia badań. Tabela zawiera również wersje testowanych narzędzi oraz wersje narzędzi do przeprowadzenia testów.

Tabela 1: Środowisko testowe – sprzęt i system

Sprzęt i system	
Procesor	Intel® Core™ i5-8300H
Pamięć RAM	16 GB DDR4 2444 MHz
Karta sieciowa	Realtek PCIe GbE Family Controller
System operacyjny	Ubuntu 20.04

Tabela 2: Środowisko testowe – narzędzia

Narzędzia	
Kubernetes	1.2.2
Docker Swarm	1.2.4
Nomad	1.2.0
Spring Boot	2.5.6
Gatling	3.5.1

4.3. Pomiar czasu startu aplikacji

W pierwszym scenariuszu czas startu aplikacji był mierzony od momentu zaczęcia tworzenia kontenerów dla aplikacji poprzez orkiestratora do czasu osiągnięcia przez aplikacje gotowości do działania wewnątrz kontenerów. Aplikacja, użyta także w kolejnych scenariuszach, uruchamiana jako aplikacja wewnątrz kontenerów została utworzona za pomocą szkieletu aplikacji Spring Boot. Listing 1. pokazuje fragment kodu zawierający kontroler REST, umożliwiający odbieranie żądań http typu GET pod adresem końcowym /, zwracając w odpowiedzi tekst *It works*.

Listing 1: Kod aplikacji testowej, używanej wewnątrz kontenerów

```
@SpringBootApplication
@RestController
public class TestApplication {
        @GetMapping(value = "/")
        public ResponseEntity<String> health() {
            return ResponseEntity.ok("It works!");
        }
        public static void main(String[] args) {
            SpringApplication.run(TestApplication.class, args);
        }
}
```

Pomiary startu tworzenia kontenerów uzyskano z metryk specyficznych dla każdego z narzędzi, natomiast czas startu aplikacji pozyskano z logów wewnątrz kontenera Docker.

4.4. Pomiar obciążenia podzespołów

Ważną rzeczą w dzisiejszych czasach jest to, aby aplikacje były zoptymalizowane pod kątem zużycia podzespołów. W tym scenariuszu sprawdzono jak duży narzut na użycie procesora, pamięci czy dysku ma każde z narzędzi oraz czy jest to znacząca wartość. W tym celu tego użyto specjalnych metryk dla każdych z narzędzi. Nomad udostępniał je natywnie, natomiast w przypadku Kubernetesa skorzystano z kube-state-metrics. Jest to projekt realizowany przez organizację Kubernetesa, który generuje metryki dla formatu Prometheusa na podstawie stanu aktualnych zasobów K8S [9]. W przypadku Docker Swarma skorzystano z podobnego projektu, jednakże przeznaczonego dla tej technologii – swarmprom, który również odpowiada za tworzenie metryk zgodnych z formatem Prometheusa.

4.5. Pomiar przepustowości i niezawodności

W dobie Internetu aplikacje internetowe doświadczają coraz to większego przepływu danych oraz liczby zapytań, więc bardzo ważną rzeczą jest, aby serwery posiadały wysoką przepustowość oraz były niezawodne. Oznacza to, że muszą przetwarzać dużą liczbę żądań na sekundę oraz jak najmniej z nich powinno kończyć się niepowodzeniem. W tym celu przeprowadzono badania z użyciem dziesięciu replik. Za rozprowadzenie ruchu odpowiadało narzędzie równoważenia obciążenia. W przypadku Kubernetesa był to Ingress-Nginx, dla Docker Swarma użyto wbudowanego natywnie narzędzia Ingress, a Nomad korzystał z HAProxy.

Do pomiarów czasów utworzono aplikację wykorzystującą bibliotekę Gatling. Jest to rozwiązanie open source służące do realizowania testów wydajnościowych [10]. Narzędzie to jest zaimplementowanie głównie w języku Scala. Wykonano 100 symulacji, w której zbadano ruch 900 użytkowników w ciągu 30 sekund. Listing 2 przedstawia fragment aplikacji odpowiadającej za uruchamianie symulacji.

Listing 2: Kod aplikacji wykonującej symulacje z użyciem biblioteki Gatling

```
class GetSimulation extends Simulation {
  val httpProtocol: HttpProtocolBuilder = http
    .baseUrl("http://localhost:8094")
    .header("Accept", "application/json")
    .header("Content-Type", "application/json")

  private val getRequest: HttpRequestBuilder = http("Get")
    .get("/")

  setup(
    scenario("Test performance for 500 users")
      .exec(get)
      .inject(constantUsersPerSec(500) during 10.seconds)
      .protocols(httpProtocol),
  )
}
```

4.6. Pomiar skuteczności mechanizmów autoregeneracji

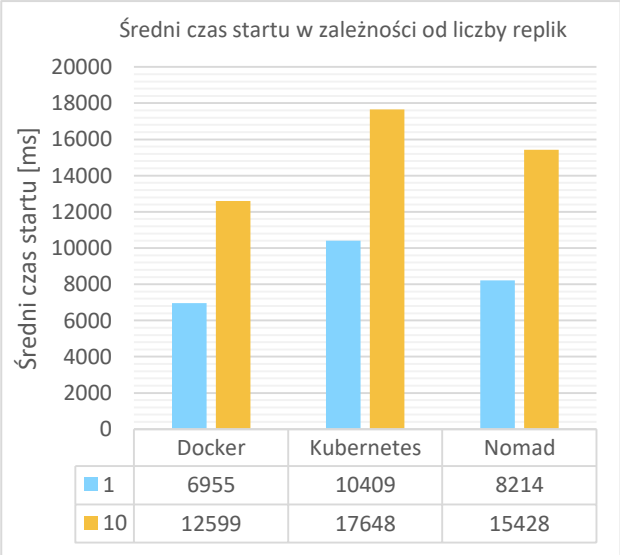
Kolejną ważną rzeczą dotyczącą systemów informatycznych jest to, aby były one projektowane

w sposób, który zapewni tolerancję potencjalnych awarii. System musi być gotowy do reakcji, gdy instancja aplikacji przestanie działać. W tym celu przygotowano badanie, które sprawdza czas działania mechanizmów autoregeneracji dla testowanych technologii. Polegało to na zmierzeniu czasu między zasymulowaniem awarii jednej z replik, czyli jej zatrzymania do czasu, kiedy nowa, uruchomiona w zastępstwie replika jest w pełni gotowa do działania. Czyli aplikacja wewnątrz kontenera potrafi przyjmować żądania i na nie odpowiadać. Testy przeprowadzono, gdy uruchomione było 10 replik aplikacji.

5. Wyniki badań

5.1. Pomiar czasu startu aplikacji

Na rysunku 1 przedstawiono średnie czasy startu aplikacji dla każdego z narzędzi, przedstawiają one pomiary przeprowadzone odpowiednio dla jednej oraz dziesięciu replik. Przeprowadzono sto takich pomiarów dla obu przypadków, a następnie wyciągnięto z nich średnią. Wykres z rysunku pierwszego pokazuje, ile czasu zajęło utworzenie aplikacji z jedną repliką. Pozwala to zaobserwować narzut infrastruktury każdego z narzędzi na uruchomienie pojedynczej aplikacji. Czasy zostały podane w milisekundach, aby zauważyć dokładniejsze różnice między badanymi technologiami.



Rysunek 1: Średnie czasy startu aplikacji testowej.

Jak można zauważyć Docker Swarm osiągnął najlepsze wyniki dla obu pomiarów. Zawdzięcza to swojej małej infrastrukturze oraz temu, że jest natywnym narzędziem silnika Docker, co pozwala na lepsze dostosowanie się do startu aplikacji umieszczonych w kontenerach. Wyniki Nomada były gorsze jedynie o 18% w przypadku jednej repliki i o 22% w przypadku dziesięciu replik. Kubernetes uzyskał najgorsze rezultaty, aczkolwiek jest to spowodowane najpotężniejszą infrastrukturą spośród narzędzi.

Tabela 3: Odchylenie standardowe i mediana czasu startu aplikacji dla jednej repliki

	Odchylenie stand. [ms]	Mediana [ms]
Docker Swarm	930,39	6 917,00
Kubernetes	882,49	10 694,00
Nomad	1 064,75	8 149,00

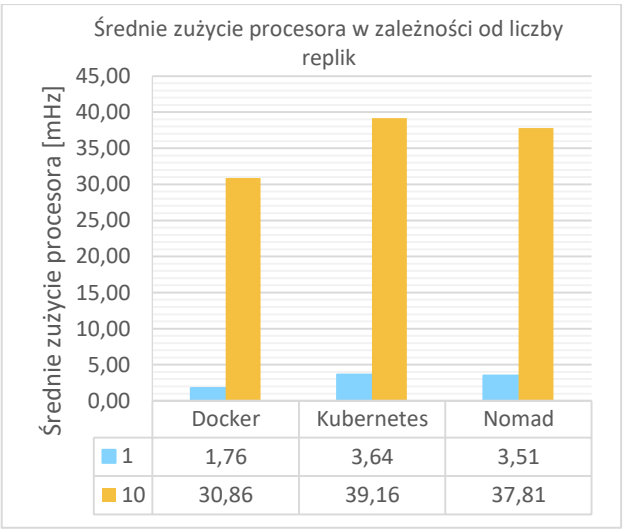
Tabela 4: Odchylenie standardowe i mediana czasu startu aplikacji dla dziesięciu replik

	Odchylenie stand. [ms]	Mediana [ms]
Docker Swarm	456,88	12 558,00
Kubernetes	1 051,78	17 637,00
Nomad	1 236,14	15 307,00

Tabele 3 i 4 przedstawiają odchylenia standardowe i mediany czasu startu aplikacji dla odpowiednio jednej i dziesięciu replik. W obu przypadkach zauważalne bardzo zbliżone do siebie miary średniej i mediany. Świadczy to o stabilnych czasach startu każdego z narzędzi. Odchylenia standardowe potwierdzają to pokazując niski i porównywalny rozrzut wyników.

5.2. Pomiar obciążenia podzespołów

Kolejnym pomiarem było zmierzenie obciążenia podzespołów przez aplikacje uruchomione za pomocą testowanych orkiestratorów. W przypadku obciążenia procesora wyniki podano w mHz, a w przypadku zużycia pomięci podano je w MB. Tak jak w poprzednim teście, badanie wykonano dla liczby jednej i dziesięciu replik i powtórzone 100 razy dla każdej z nich. W pierwszym przypadku pomogło to sprawdzić, czy narzędzia potrafią przyznawać odpowiednią ilość zasobów uruchomionej aplikacji. Drugi przypadek pozwolił na sprawdzenie czy orkiestratory są wstanie zoptymalizować zużycie podzespołów, gdy uruchomione jest kilka instancji tej samej aplikacji.



Rysunek 2: Średnie obciążenia procesora przez testową aplikację.

Rysunek 2 przedstawia wykresy ilustrujące średnie obciążenie procesora dla odpowiednio jednej i dziesięciu replik aplikacji testowej. W pierwszym przypadku widać, że Docker Swarm uzyskał wyniki ponad dwa razy niższe od pozostałych testowanych narzędzi. Pokazało to, że Docker Swarm poprzez bycie natywnym rozwiązaniem silnika Docker dobrze zoptymalizował utworzenie pojedynczego kontenera i narzut infrastruktury był najmniejszy. Drugi przypadek pokazuje już mniejsze różnice. Widać, że w porównaniu do jednej repliki aplikacja zreplikowana dziesięciokrotnie obciążyła procesor o wiele bardziej w przypadku Swarma. Natomiast wyniki Kubernetesa i Nomada pokazały, że obciążenie rośnie proporcjonalnie wraz z ilością replik. Jest to pozytywną informacją w przypadku dużych systemów, w których liczby replik potrafią sięgać setek, co pozwala na oszacowanie potencjalnego obciążenia procesora maszyny serwerowej. Zauważalne są tutaj zalety dla małych systemów, które nie posiadają wielu replik. Spowodowane jest to bardzo dobrym zarządzaniem zużyciem procesora dla małej liczby podów przez Docker Swarm.

Tabele 5 i 6 zawierają miary odchylenia standardowego i mediany dla zużycia procesora z użyciem odpowiednio jednej i dziesięciu replik. Odchylenie od wartości przeciętnej jest pomijalne. Pokazuje to stabilność zarządzania użyciem procesora przez narzędzia. Mediany dla obu przypadków są porównywalne do miar średnich, a w przypadku Kubernetesa dla dziesięciu replik równa średniej.

Tabela 5: Odchylenie standardowe i mediana zużycia procesora dla jednej repliki

	Odchylenie stand. [mHz]	Mediana [mHz]
Docker Swarm	0,15	1,74
Kubernetes	0,13	3,65
Nomad	0,14	3,47

Tabela 6: Odchylenie standardowe i mediana zużycia procesora dla dziesięciu replik

	Odchylenie stand. [mHz]	Mediana [mHz]
Docker Swarm	1,06	30,82
Kubernetes	0,55	39,16
Nomad	2,18	38,14

Tabele 7 i 8 przedstawia odchylenie standardowe i medianę dla zużycia pamięci przez uruchomione aplikacje. Istotną informacją, którą można odczytać z tabel jest pomijalne odchylenie.

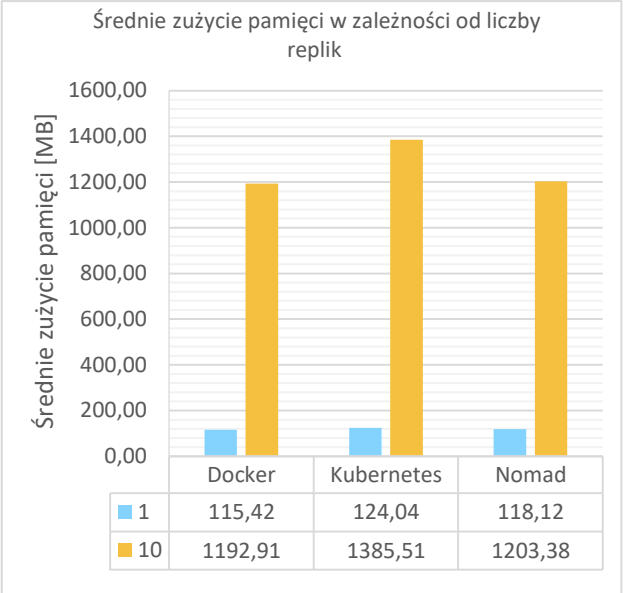
Rysunek 3 ilustruje średnie zużycie pamięci ponownie dla jednej i dziesięciu replik. Widać, że na pierwszym wykresie wyniki są niemal identyczne dla wszystkich narzędzi i każdy z orkiestratorów podobnie radzi sobie z zarządzaniem pamięcią dla kontenerów, które uruchamia. Analogicznie dla dziesięciu replik wyniki były znów niemal identyczne i nie zauważono znaczących optymalizacji, gdy zwiększono liczbę replik tej samej aplikacji. Wyniki były dziesięć razy większe niż dla pojedynczej replik. Oznacza to, że każda replika zużywała tyle samo pamięci co w przypadku pojedynczego uruchomienia aplikacji.

Tabela 7: Odchylenie standardowe i mediana zużycia pamięci dla jednej repliki

	Odchylenie stand. [MB]	Mediana [MB]
Docker Swarm	4,54	115,00
Kubernetes	6,87	123,00
Nomad	5,73	118,00

Tabela 8: Odchylenie standardowe i mediana zużycia pamięci dla dziesięciu replik

	Odchylenie stand. [MB]	Mediana [MB]
Docker Swarm	51,79	1 189,00
Kubernetes	31,86	1 385,00
Nomad	44,18	1 206,00



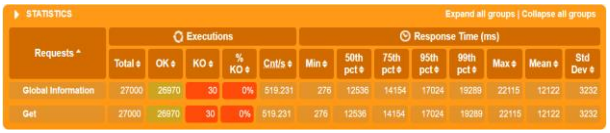
Rysunek 3: Średnie zużycie pamięci przez testową aplikację.

Podsumowując każde z narzędzi dobrze sprawdziło się w aspekcie zarządzania pamięci orkiestrowanych aplikacji. Jedynie w przypadku dziesięciu replik narzut Kubernetesa wzrósł o nieznaczącą wartość, co mimo wszystko nie powoduje różnic. Żadne z narzędzi nie uzyskało tutaj przewagi, co pokazuje, że coraz trudniej optymalizować aplikacje pod względem zużycia pamięci i programiści muszą tworzyć przemyślane systemy.

5.3. Pomiar przepustowości i niezawodności

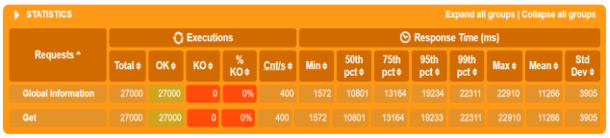
Kolejnym testem było sprawdzenie przepustowości i niezawodności narzędzi, które współpracują z mechanizmami odpowiedzialnymi za równoważenia obciążenia. W tym przypadku stworzono 10 replik

aplikacji testowej. Rysunki 4, 5 i 6 przedstawiają fragment raportu wygenerowanego przez narzędzie Gatling po wykonaniu testu obciążeniowego dla każdego z orkiestratorów.



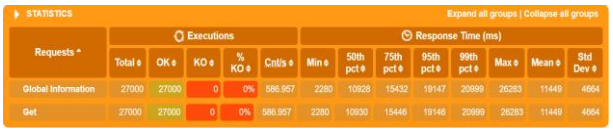
Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	27000	26970	30	0%	519.231	276	12536	14154	17024	19289	22115	12122	3232
Get	27000	26970	30	0%	519.231	276	12536	14154	17024	19289	22115	12122	3232

Rysunek 4: Fragment tabeli z raportu wygenerowanego przez narzędzie Gatling dla Docker Swarm.



Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	27000	27000	0	0%	400	1572	10801	13164	19234	22311	22910	11266	3905
Get	27000	27000	0	0%	400	1572	10801	13164	19233	22311	22910	11266	3905

Rysunek 5: Fragment tabeli z raportu wygenerowanego przez narzędzie Gatling dla Kubernetes.



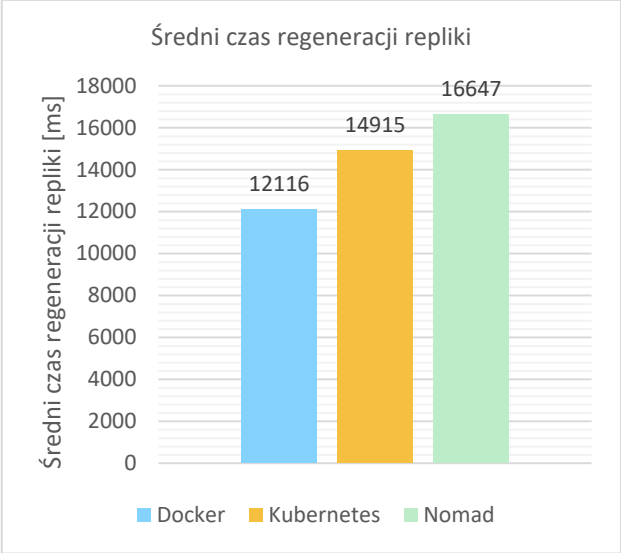
Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	27000	27000	0	0%	586.957	2280	10928	15432	19147	20999	26283	11449	4664
Get	27000	27000	0	0%	586.957	2280	10930	15446	19146	20999	26283	11449	4664

Rysunek 6: Fragment tabeli z raportu wygenerowanego przez narzędzie Gatling dla Nomad Hashicorp.

Dla każdego przypadku wykonano sumarycznie 27000 żądań w czasie 30 sekund, co daje 900 żądań na sekundę. Tabela przedstawia informacje na temat liczby obsłużonych poprawnie żądań oraz tych które zakończyły się niepomyślnie. Ponadto tabela zawiera również dane statystyczne takie jak średnia, mediana, minimalny i maksymalny czas żądania, jak również rozkład percentyli. Percentyle przedstawiają wartości, poniżej których znajduje się określony procent danych. Dla powyższego przykładu 99-ty percentyl to czas odpowiedzi, dla którego 99% przypadków osiągał krótszy czas odpowiedzi serwera. Dzięki przedstawionym wynikom widać, że największy minimalny i maksymalny czas odpowiedzi uzyskał Nomad, a mimo to zaliczył najwyższy współczynnik zapytań na sekundę. Średnie czasy odpowiedzi były niemal identyczne dla każdego z narzędzi. Podobna sytuacja ma miejsce w przypadku rozkładu percentyli. Są to wartości w większości takie same dla wszystkich orkiestratorów, różnią się pojedynczymi wynikami np. dla 95-tego percentyla na korzyść Docker Swarma lub 50-tego percentyla na jego niekorzyść. Ważną rzeczą, którą można zauważyć jest to, że 30 zapytań dla systemu orkiestracyjnego Dockera zakończyło się niepowodzeniem. Z uwagi na problemy z połączeniem doszło do przekroczenia maksymalnego, dozwolonego czasu. Jest to poważny minus, ponieważ mimo podobnych wyników czasowych, Docker Swarm nie przetworzył wszystkich zapytań, co w dobie dzisiejszych systemów jest dużą wadą, mimo że to jedynie 0.1% wszystkich zapytań.

5.4. Pomiar skuteczności mechanizmów autoregeneracji

Ostatnim badaniem było sprawdzenie skuteczności i sprawności mechanizmów autoregeneracji testowanych narzędzi. W tym celu znów utworzono 10 replik aplikacji testowej, a następnie usunięto jedną z nich i zmierzono czas, w którym każdy z orkiestratorów uruchomił nową instancję aplikacji, gotową do działania, czyli zdolną do przyjęcia żądań. Rysunek 7 przedstawia średni czas ze stu pomiarów dla każdego z narzędzi.



Rysunek 7: Średni czas regeneracji repliki dla narzędzi.

Jak widać na rysunku 7 różnice między wynikami nie są bardzo duże, jednak najlepszy rezultat uzyskał Docker Swarm. Znaczenie miało to, że jest to natywne rozwiązanie do orkiestracji silnika Docker, dzięki czemu przywrócenie do życia pojedynczego kontenera zostało zrealizowane najszybciej. Na drugim miejscu znalazł się Kubernetes, a nieznacznie gorszy wynik uzyskał Nomad.

Tabela 9: Odchylenie standardowe i mediana czasu regeneracji repliki

	Odchylenie stand. [ms]	Mediana [ms]
Docker Swarm	841,68	12 176,00
Kubernetes	631,84	14 886,00
Nomad	574,28	16 634,00

Tabela 9 przedstawia odchylenie standardowego i medianę dla czasów regeneracji repliki. Otrzymane wyniki wskazują na jednolite czasy uzyskane w trakcie badań. Potwierdza to stosunkowo małe odchylenie oraz mediana, której miara jest zbliżona do miary średniej. Oznacza to stabilność mechanizmów przywracania, dzięki czemu programiści mogą przewidzieć, ile czasu zajmie potencjalne naprawienie awarii.

Dodatkowym narzutem dla obu technologii było to, że narzędzia nie korzystają z natywnych rozwiązań do równoważenia obciążenia. Przywróconą do życia replikę należało ponownie dodać do odpowiednio Nginxa dla Kubernetesa oraz HAProxy dla Nomada, co mogło nieznacznie wpłynąć na uzyskany rezultat końcowy. Mimo to mechanizm autoregeneracji dla

każdego z orkiestratorów zadziałał poprawnie, czyli przywrócił uszkodzoną replikę.

6. Wnioski

Trudno jest podjąć optymalną decyzję podczas wyboru odpowiedniego orkiestratora, ponieważ to właśnie tego typu narzędzia odpowiadają za zarządzanie całym naszym systemem. Niniejsza praca, skupiła się na porównaniu wydajności i funkcjonalności trzech narzędzi służących do orkiestracji kontenerów. Miało to na celu ułatwienie podjęcia decyzji dotyczącej wyboru odpowiedniego narzędzia dostosowanego do systemu. W analizie porównawczej wzięto pod uwagę kryteria takie jak: czas startu aplikacji, obciążenie podzespołów, niezawodność działania i przepustowość oraz czas autoregeneracji.

Biorąc pod uwagę kryterium czasu startu aplikacji Docker Swarm uzyskał najlepsze rezultaty, szczególnie dla aplikacji, które uruchamiane są jako pojedyncze repliki. Duży wpływ na to ma wcześniej wspomniany fakt, że Swarm jest natywnym narzędziem silnika Docker.

Porównując wyniki dotyczące obciążenia podzespołów przez aplikacje, którymi zarządzały orkiestratory widać znów dużą przewagę Swarma w aspekcie użycia procesora dla pojedynczej repliki. Natomiast przy większej liczbie replik różnice zacierają się. Na korzyść Nomada i Kubernetesa przemawia fakt, że zużycie pamięci rośnie proporcjonalnie dla kolejnych replik, natomiast w przypadku natywnego narzędzia do wsparcia trybu klastrowania Dockera, zużycie dla większej liczby replik wzrosło znacząco.

Pod względem przepustowości i niezawodności można zauważyć, że biorąc pod uwagę średni czas odpowiedzi na zapytanie przez każde z narzędzi, orkiestratory zrealizowały zadanie niemal identycznie. Jednakże Kubernetes osiągnął najgorszy wynik, jeśli chodzi o liczbę zapytań na sekundę, a najlepiej wypadł Nomad, choć otrzymał on największy maksymalny wynik odpowiedzi. Największym problemem Docker Swarma okazało się jednak to, że pewna część zapytań zakończyła się niepowodzeniem, ponieważ został przekroczony maksymalny czas oczekiwania, co nie zdarzyło się w przypadku pozostałych dwóch narzędzi. Jest to duża wada, ponieważ w przypadku niektórych systemów jest niedopuszczalnym, aby serwis nie był gotowy przetworzyć żądań.

W ostatnim badaniu dotyczącym mechanizmów autoregeneracji Swarm uzyskał najkrótszy czas uruchomienia nowej aplikacji w miejsce uszkodzonej repliki. Na drugim miejscu uplasował się Kubernetes, a ostatnie miejsce zajął Nomad. Jednakże czasy nie różniły się znacząco, a wszystkie narzędzia dobrze poradziły sobie z tym zadaniem, więc żadne z nich nie uzyskało znaczącej przewagi w tym aspekcie.

Z przeprowadzonej analizy można wysnuć wniosek, że najlepszym narzędziem orkiestracyjnym z trzech badanych orkiestratorów jest Docker Swarm. Jednakże nie zostały porównane funkcjonalności każdego z narzędzi, które w przypadku Kubernetesa i Nomada są o wiele większe niż możliwości Docker Swarm. W celu dokonania dokładniejszej analizy i oceny narzędzi trzeba by uwzględnić kolejne aspekty oraz przeprowadzić badania na różnorodnych aplikacjach.

Literatura

[1] R. Dua, A. Raja, D. Kakadia, Virtualization vs containerization to support paas, 2014 IEEE International Conference on Cloud Engineering (2014) 610-614.

[2] B. Rad, H. Bhatti, M. Ahmadi, An introduction to docker and analysis of its performance, International Journal of Computer Science and Network Security 17(3) (2017) 228.

[3] M. Moravcik, M. Kontsek, Overview of Docker container orchestration tools, 2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA) (2020) 475-480.

[4] I. Al Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli, Container orchestration engines: A thorough functional and performance comparison, ICC 2019-2019 IEEE International Conference on Communications (ICC) (2019) 1-6.

[5] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, R. Sinnott, A performance comparison of cloud-based container orchestration tools, 2019 IEEE International Conference on Big Knowledge (ICBK) (2019) 191-198.

[6] What is Kubernetes?, https://kubernetes.io/pl/docs/concepts/overview/what-is-kubernetes/. , [30.01.2022]

[7] P. Pedamkar, What is Docker Swarm?, https://www.educba.com/what-is-docker-swarm/ , [02.02.2022]

[8] Introduction to Nomad, https://learn.hashicorp.com/tutorials/nomad/get-started-intro?in=nomad/get-started , [02.02.2022]

[9] kube-state-metrics – Introduction to Kubernetes metrics, https://kubernetes.io/blog/2021/04/13/kube-state-metrics-v-2-0/ , [02.02.2022]

[10] What is Gatling?, https://gatling.io/open-source/, [08.02.2022]