WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

Wirtualizacja systemów IT

Projekt *Dockery – konteneryzacja (rzetelny opis i wykonanie praktycznego doświadczenia)*

inż. Hubert Makowski

Grupa: WCY24KX1S4

Prowadzący: mgr inż. Łukasz Paciorek

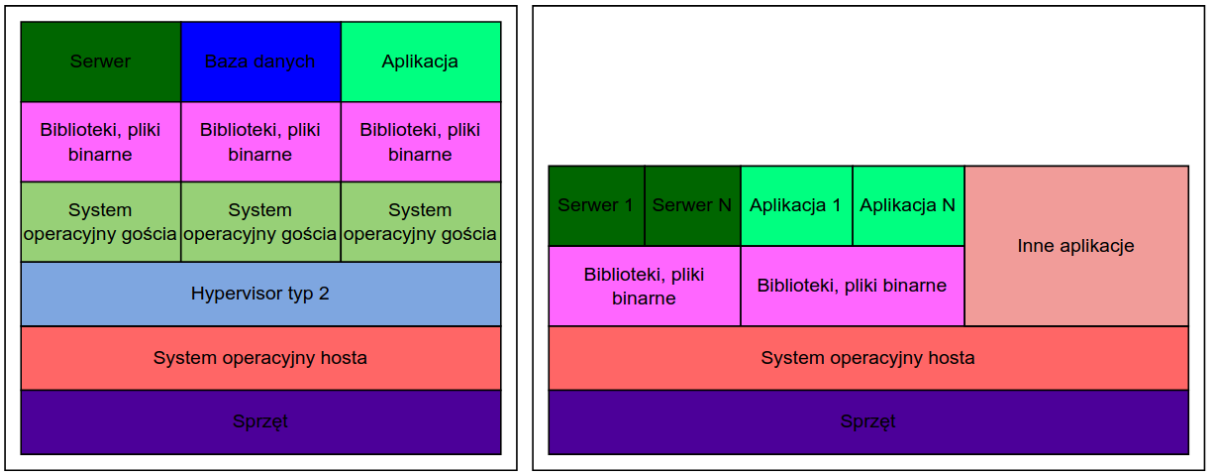
Spis treści

Wprowadzenie do konteneryzacji:

Docker to otwarta platforma, która umożliwia tworzenie, dystrybucję oraz uruchamianie aplikacji w lekkich, izolowanych środowiskach zwanych kontenerami. Dzięki separacji aplikacji od infrastruktury[1]. Docker pozwala na szybsze i bardziej spójne wdrażanie oprogramowania, umożliwiając programistom pracę w standaryzowanych środowiskach niezależnych od konfiguracji systemu operacyjnego hosta. Platforma wykorzystuje architekturę klient-serwer, w której klient komunikuje się z demonem Docker odpowiedzialnym za zarządzanie obrazami, kontenerami, sieciami i wolumenami, co ułatwia budowanie, testowanie oraz skalowanie aplikacji w różnych środowiskach – od lokalnych laptopów, przez centra danych, aż po chmury hybrydowe.

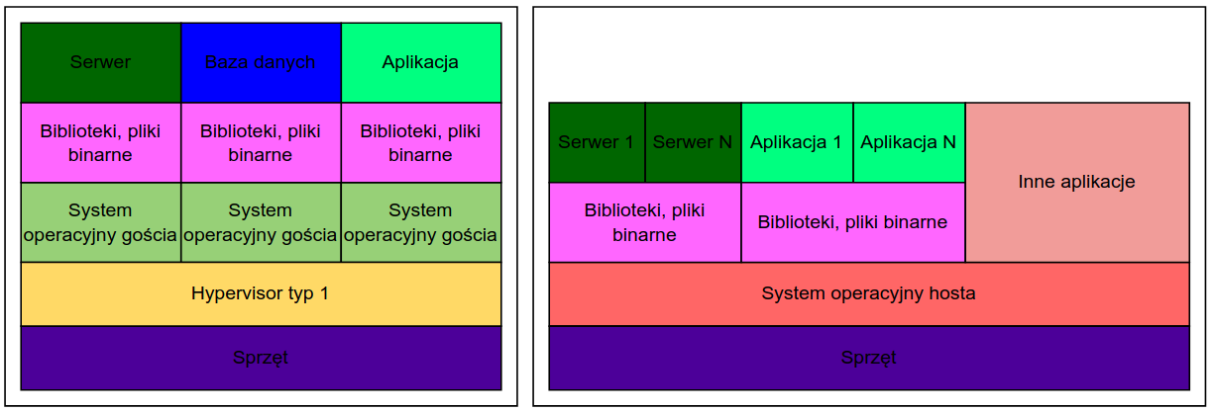
Technika wirtualizacji nie pozwala osiągnąć pełnej przenośności. Ewolucja konteneryzacji pozwala uzyskać przenośność aplikacji [2]. Kontener zawiera aplikację, a także niezbędne biblioteki, pliki binarne i inne elementy zależne. Komponenty te są zagregowane. Kompletny i kompaktowy kontener można dostarczyć klientowi, a następnie uruchomiać, zarządzać wdrażać. Ekosystem Docker dąży do osiągniecia maksymalnej automatyzacji. Kontenery spełniają założenia agile i DevOps - pozwalają na tworzenie zwinnych i niedrogich aplikacji.

Aplikacje uruchamiane w maszynie wirtualnej są odseparowane od systemu operacyjnego hosta za pomocą hiper nadzorcy typu 2. Kontener Dockera osiąga izolację dzięki możliwościom jądra Linuksa. W przypadku wirtualizacji typu 1 hiper nadzorca zapewnia funkcje systemu operacyjnego, a także funkcje monitorowania i zarządzania. Poniższe obrazy przedstawiają narzut programowy wynikający ze stosowania hiper nadzorcy w porównaniu z kontenerami. Narzut jest większy dla wirtualizacji typu drugiego, co skutkuje spadkiem wydajności. Gdy proces jest uruchamiany wewnątrz kontenera, dodawany jest tylko niewielki kod wewnątrz jądra do zarządzania kontenerem. W maszynie wirtualnej wywołania z procesu do sprzętu lub hiper nadzorcy wymagają dwukrotnego przełączania do trybu chronionego, co zmniejsza szybkość wywołań.



Rys. 1. Wirtualizacja typu 2 i konteneryzacja

Źródło: Opracowanie własne na podstawie [2]



Rys. 2. Wirtualizacja typu 1 i konteneryzacja

Źródło: Opracowanie własne na podstawie [2]

Kontenery zajmują mniej przestrzeni na dysku, nie zawierają jądra systemu, są uruchamiane i konfigurowane szybciej niż maszyny wirtualne. Kontenery są skierowane na aplikacje oparte o mikrousługi i pozwalają oszczędzać zasoby sprzętowe.

Izolacja

Kontener Docker działa jako aplikacja hosta i podlega standardowym zabezpieczeniom oferowanym przez system operacyjny. Docker obsługuje możliwości oferowane przez jądro Linuks - przestrzenie nazw oraz zbiory procesów. Można wyróżnić pięć przestrzeni, które oferują globalną izolację zasobów systemowych:

1. sieć - izolowanie zasobów sieciowych
2. udostępnienia - izolowanie udostępnienia systemu plików
3. PID - izolowanie id procesu
4. użytkownik - izolowanie id użytkownika i id grupy
5. UTS - izolowanie nazwy hosta i domeny NIS

Szkielet Docker korzysta z hierarchicznej przestrzeni procesów, procesy uruchamiane wewnątrz kontenera nie mają dostępu do procesu nadrzędnego. Sieciowa przestrzeń nazwa pozwala na tworzenie niezależnych interfejsów sieciowych dla każdego kontenera. Przestrzeń nazw udostępnienia zapewnia, że system plików jest dostępny tylko dla procesu należącego do tej samej przestrzeni nazw. Przestrzeń nazw użytkownika pozwala przyznać uprawnienia administratora wewnątrz określonej przestrzeni nazw.

Kontenery są oparte na grupach kontrolnych cgroups, które śledzą oraz odsłaniają statystyki obciążenia zasobów systemu. Grupy kontrolne ograniczają przydział zasobów systemowych dla kontenerów Docker. Polecenia Dockera wspierają ograniczenie dotyczące procesora, pamięci, przestrzeni wymiany i operacji odczytu/zapisu podczas uruchamiania kontenera. Moc CPU systemu jest opisywana liczbą 2^{10}. Konfigurując udziały CPU kontenera można ustalić ile czasu CPU może otrzymać kontener. Nie są to udziały na wyłączność, a raczej podpowiedź dla kolejkowania procesów. Kontener można podpiąć do kilku rdzeni CPU. Przez to zadanie kontenera będą wykonywane tylko na przypisanych rdzeniach. Można kontrolować zakres pamięci do jakiej kontenera dostęp, limit pamięci jest twardy

Bezpieczeństwo kontenerów Docker

Firma Docker Inc. wrodziła otwarty format obrazów umożliwiających tworzenie pakietów aplikacji. Uruchamianie aplikacji w jednym systemie naraża je na:

1. wykorzystanie jądra systemu hosta - błąd jądra hosta może doprowadzić do wydostania się na zewnątrz procesu kontenera
2. ataki DoS - wszystkie kontenery współdzielą zasoby jądra. Jeśli jeden proces byłby w stanie zmonopolować zasoby jądra, inne procesy kontenera nie byłyby w stanie świadczyć poprawnie usług
3. wyłamanie poza kontener - proces, który wyłamie się poza kontener będzie miał takie same przywileje jakby działał w kontenerze. Proces kontenera o wysokich uprawnieniach będzie posiadał wysokie uprawnienia w systemie hosta
4. zatrute obrazy - modyfikacja obrazów Dockera może wpłynąć na działanie kontenera i hosta.

Podstawowe zabezpieczenia demona Dockera to:

1. szyfrowanie ruchu - TLS pozwoli na szyfrowanie ruchu do serwera Dockera na port 2376
2. uwierzytelnianie użytkowników - wyłącznie poprzez certyfikaty, można wykorzystać centrum autoryzacji

Porównanie maszyny wirtualnej i kontenerów dockera pod względem bezpieczeństwa

Bezpieczeństwo maszyn wirtualnych i kontenerów to sprawa dyskusyjna. Istnieją argumenty przemawiające za obiema stronami. Każda próba uzyskania dostępu przez maszynę wirtualną musi przejść przez hiper nadzorcę. Pole manewru podczas ataku jest mniejsze dla maszyny wirtualnej. Przez uzyskaniem dostępu do maszyny hosta haker musi uzyskać dostęp do jądra maszyny wirtualnej oraz hiper nadzorcy. Kontenery działają na jądrze hosta. Izolacja jest implementowana w ramach pojedynczego jądra, jest to wirtualizacja systemu operacyjnego[3]. Mniejsza liczba warstw technologii ułatwia dostęp do hosta, przez co kontenery co bardziej narażone na ataki. Platforma Docker umożliwia odizolowanie aplikacji od siebie, odizolowanie aplikacji od hosta, ograniczenie możliwości aplikacji, stosowanie zasady najniższych przywilejów. Wszelkie próby modyfikacji kontenerów spowodują zawieszeni działania, ponieważ kontenery są niemodyfikowalne. Zmniejsza to wektor ataków poprzez mutacje. Grupy cgroups zapewniają, że żaden z procesów kontenera nie przejmie zasobów systemu. Rozwiązanie to pomaga zapobiegać ataków DoS. Domyślnie Docker Engine uruchamia kontenery z uprawnieniami administratora (z których korzysta aplikacja działająca na kontenerze). W kontenerze można udostępnić dowolny katalog hosta. Można temu zapobiec używając odpowiedniego parametru przy uruchamianiu kontenera.

Zestaw modyfikacji systemu Linuks SELinuks ma załatać dziury zabezpieczeń kontenerów. Implementuje Mandatory Access Control, Multi-Level Security, Multi-Category Security. Kontenery Dockera są automatycznie przypisywane do kontekstu SELinuksa. SELinuks sprawdza dozwolone operacje po pełnym sprawdzeniu dyskretnej kontroli dostępu. Kontrola MAC wymusza administracyjne zdefiniowanie zasad bezpieczeństwa dla wszystkich procesów i plików w systemie. MAC zapewni silną separację aplikacji. MAC pozwala na zatrzymanie działa każdego użytkownika (nawet root) przed operacją na zasobie do którego nie powinien mieć dostępu. MCS zapewnia ochronę kontenerów przed innymi kontenerami. Jądro systemu umożliwia pracę na zasobach kontenerowi tylko o tej samej etykiecie(liczba losowa). Zhakowany proces kontenera musiałby znać etykietę (1024 bity) żeby wpłynąć na inne kontenery. Kolejne zabezpieczenie SCONE oferuje interfejs, który szyfruje i odszyfrowuje dane wejścia/wyjścia. Mechanizm korzysta z funkcji zaufane wykonywania SGX obłuskiwane przez CPU Intel. Docker Container Trust (DCT) zabezpiecza przed fałszowaniem obrazów oraz zapewnia odporność na ataki MiTM w sieci. DCT chroni przed atakami powtórzeniowymi korzystając z klucza znaku czasu tworzonego w momencie publikowania obrazu.

Największa niechlujnością administratora kontenerów jest niekorzystanie z trybu bez uprawnień roota lub mechanizmu users-remap w demonie Dockera. Obejście mechanizmów przestrzeni nazw może pozwolić userowi root kontenera działanie na

zasobach hosta Dockera, tak jak user root hosta. Bezwzględnie należy mieć na uwadze zamontowane katalogi do serwer Dockera. Mając zamontowany katalog /etc użytkownik root kontenera może dokonać dowolnych operacji na katalogu np. czytanie /etc/shadow. Jednym ze sposobów mitygacji jest uruchamianie kontenerów z innymi UID niż używa hosta Dockera. Kolejny sposób polega na uruchamianiu kontenerów przez nieuprzywilejowanego użytkownika. Możliwe jest też uruchomienie demona Dockera bez uprawnień roota. Wiąże się to jednak z konsekwencjami. Część kontenerów musi zostać uruchomiona z uprawnieniami root, a nie jest to możliwe gdy demon Dockera nie został uruchomiony z uprawnieniami roota.

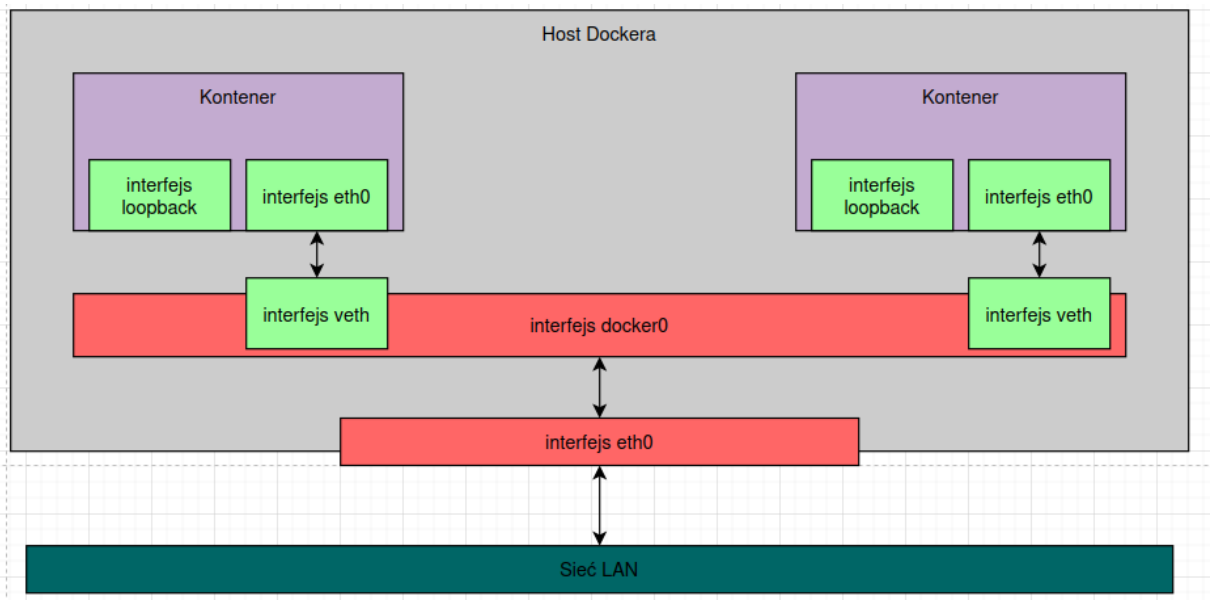
Zarządzanie danymi

Czas życia środowiska aplikacji działającej w kontenerze jest równoważne czasu życia kontenera. Dotyczy to również danych składowanych przez aplikacje. Podczas podnoszenia wersji kontenera, usuwania kontenera pliki aplikacji muszą zostać wyciągnięte i przekazane między kontenerami. Docker oferuję dostęp do plików kontenera poprzez wolumin danych. Docker umożliwia składowanie danych na trzy sposoby: mechanizm zarządzania woluminami Dockera, podpięcie katalogu hosta Dockera do lokalizacji wewnątrz kontenera, stworzenie specjalnego kontenera, który udostępnia i przechowuje dane. Wolumin Docker jest częścią systemu plików kontenera. Docker Engine tworzy automatycznie wolumin w katalogu /var/lib/docker/volumes/. Katalog lub plik hosta może zostać udostępniony jako wolumin tylko podczas uruchomienia kontenera. Kontenery zawierające tylko dane są nazywane w sposób jawny, przez co inne kontenery mogą odwołać się do nazwy kontenera danych. Woluminy kontenera danych są dostępna nawet, gdy kontener danych jest zatrzymany. Wolumin danych kontenera może zostać udostępniony w kilku kontenerach.

Obsługa sieci

Sieć jest niezbędnym elementem infrastruktury rozwiązań biznesowych. Pierwsze wersje Dockera obsługiwały tylko mostek sieciowy, później wprowadzono technologie SDN firmy SocketPlane. Podczas instalacji Docker silnik tworzy sieci bridge, host i null. W momencie tworzenia nowego kontenera generowany jest stos sieci oraz połączenie miedzy kontenerem i siecią bridge. Wybranie sieci host spowoduje, że kontener będzie współdzielił adres IP i port hosta. Wybranie sieci null tworzy tylko interfejs pętli zwrotnej w kontenerze. Sieć bridge jest domyślnie przypisywana kontenerowi. Docker tworzy interfejs mostka Ethernet wewnątrz jądra Linuksa w hoście Dockera. Interfejs mostka docker0 służy do przesyłania ramek między kontenerami oraz kontenerami i siecią zewnętrzną. Interfejs eth0 kontenera jest połączony z mostkiem docker0 za pomocą interfejsu veth. Interfejsy veth i eth0 należą do interfejsu sieciowego Virtual Ethernet. Interfejs veth jest składa się z dwóch interfejsów - jeden jest przypisany do kontenera

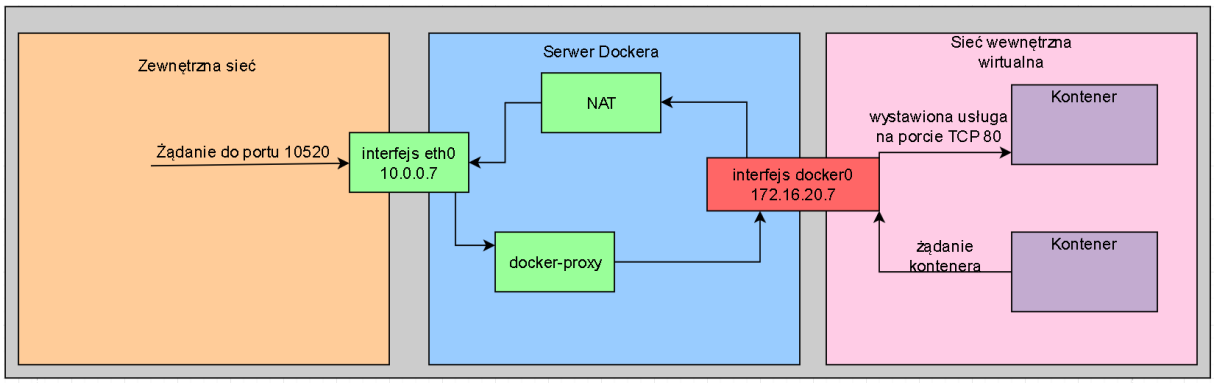
(eth0), drugi jest związany z mostkiem docker0. Docker przypisuje adres IP do kontenera, który jest nieosiągalny poza hostem Dockera. Obrazuje to rysunek poniżej.



Rys. 3. Schemat sieci Docker

Źródło: Opracowanie własne na podstawie [2]

Docker umożliwia publikowanie usługi działającej w kontenerze poprzez powiązanie portu kontenera z interfejsem hosta. Ramki kierowane do adresu IP i portu hosta Dockera będą przekierowywane do portu kontenera. W przypadku pracy z zewnętrznymi obrazami musi dojść do jednoznacznego określenia portu używanego przez aplikację wewnątrz kontenera. Instrukcja EXPOSE umożliwia podanie informacji o porcie.



Rys. 4. Wystawiona usługa

Źródło: Opracowanie własne na podstawie [2]

Mechanizm obsługi logów

Logi mogą mieć krytyczne znaczenie dla przywrócenia działania aplikacji. Muszą być dobrze przygotowane. Żaden z podstawowych mechanizmów logowania Linuksa nie zadziała na Docker bez konfiguracji. Docker upraszcza obsługę logów:

1. Przechwytuje wszystkie dane tekstowe zwracane przez aplikacje. Wszystko przesyłane na standardowe wejście oraz wyjście błędu jest przesyłane do skonfigurowanego logowania backendu.
2. Wiele opcji jest dostępnych w postaci wtyczek

Standardowa metoda logowania używa plików json. Logi aplikacji przesyłane są w strumieniu przez demona Dockera do pliku JSON tworzonego dla każdego kontenera. Rzeczywiste pliki znajdują się na serwerze Dockera w katalogu /var/lib/docker/conatiners/<id_kontenera>.

Zaawansowane mechanizmy opierają się na wtyczkach, które pozwalają na przesyłanie logów do różnych frameworków i usług. Najpopularniejsza wtyczka daje możliwość przesyłania logów kontenera do sysloga. Problem pojawia się przy przesyłania logów z Dockera do zdalnego serwera logów za pomocą TCP. Przy uruchomieniu kontenera Docker próbuje połączyć się z serwerem. Jeśli połączenie się nie powiedzie, kontener nie uruchomi się. Używają UDP mamy pewność, że kontener uruchomi się, ale tracimy na szyfrowaniu i niezawodności TCP.

Monitorowanie Dockera

Bez łatwego w utrzymaniu monitorowania i kontrolowania nie można w dłuższym okresie utrzymywać systemów produkcyjnych. Docker udostępnia narzędzia pozwalające sprawdzać stan kontenera oraz daje możliwości raportowania. Polecenie docker container stats pozwala podejrzeć następujące parametry: Id kontenera użycie CPU wykorzystanie pamięci statystki sieciowe, operacji wejścia/wyjścia liczbę aktywnych procesorów Mechanizm pozwala na ustalenie limitu pamięci, który nie spowoduję zatrzymanie kontenera. Liczba aktywnych procesorów służy do debugowania, czy tworzone procesy są usuwane. Węzeł /stats/ API Dockera udostępnia statystki w formacie JSON. Większość demonów Dockera udostępnia API tylko na gnieździe UNIX. Produkcyjnie można korzystać z gniazda TCP, chociaż autor [3] nie zaleca tego podejścia. System Prometheus jest popularnych rozwiązanie do monitorowania systemów rozporoszonych. Docker udostępnia interfejs dedykowany dla Prometheus'a. Warto zaznaczyć, że rozwiązanie udostępnia statystyki serwera Dockera, a nie kontenerów. W kontrze do tego stoi produkt cAdvisor, które monitoruje kontenery. Najprościej jest uruchomić cAdvisor jako kontener Dockera.

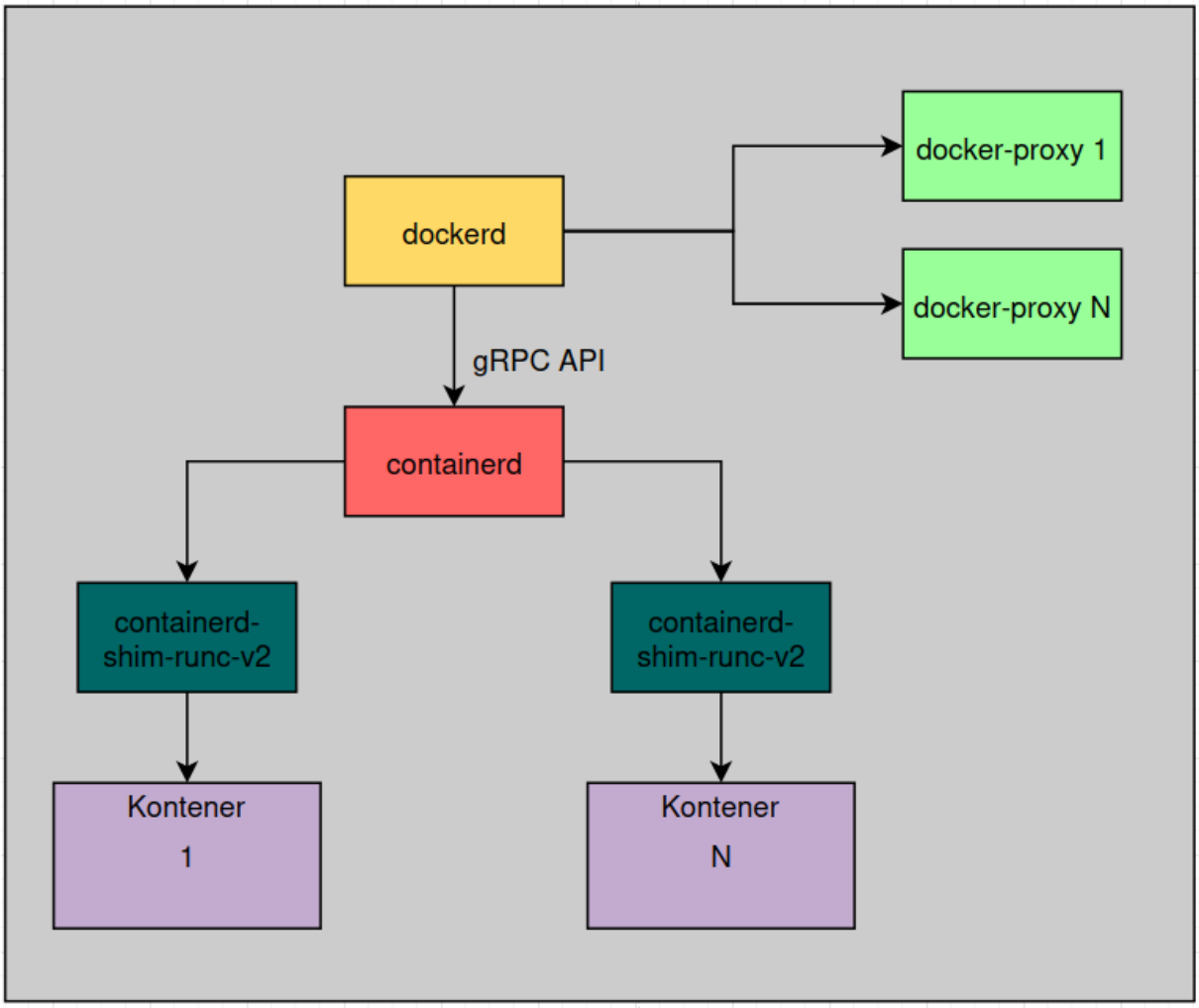
Architektura Dockera

Docker zbudowany jest z pięciu komponentów serwerowych:

1. dockerd - jeden na serwer, udostępnia API, buduje obrazy, zarządza sięcią, woluminami, logowaniem
2. containerd - jeden na serwer, zarządza niskopoziomowymi sterownikami sieci
3. runc - tworzy kontener i uruchamia go

4. conteinerd-shim-runc-v2 - jeden na kontener, obsługuje deskryptory plików przekazywane do kontenera (stdin, stdout)
5. docker-proxy - jeden na regułę przekazywania portów, przekazuje ruch z kontenera do hosta

Komponenty są widoczne jako całość poprzez API. Containerd umożliwia integrację z K8S, zamiast wymuszać korzystanie z API



Rys. 5. Architektura Dockera

Źródło: Opracowanie własne na podstawie [2]

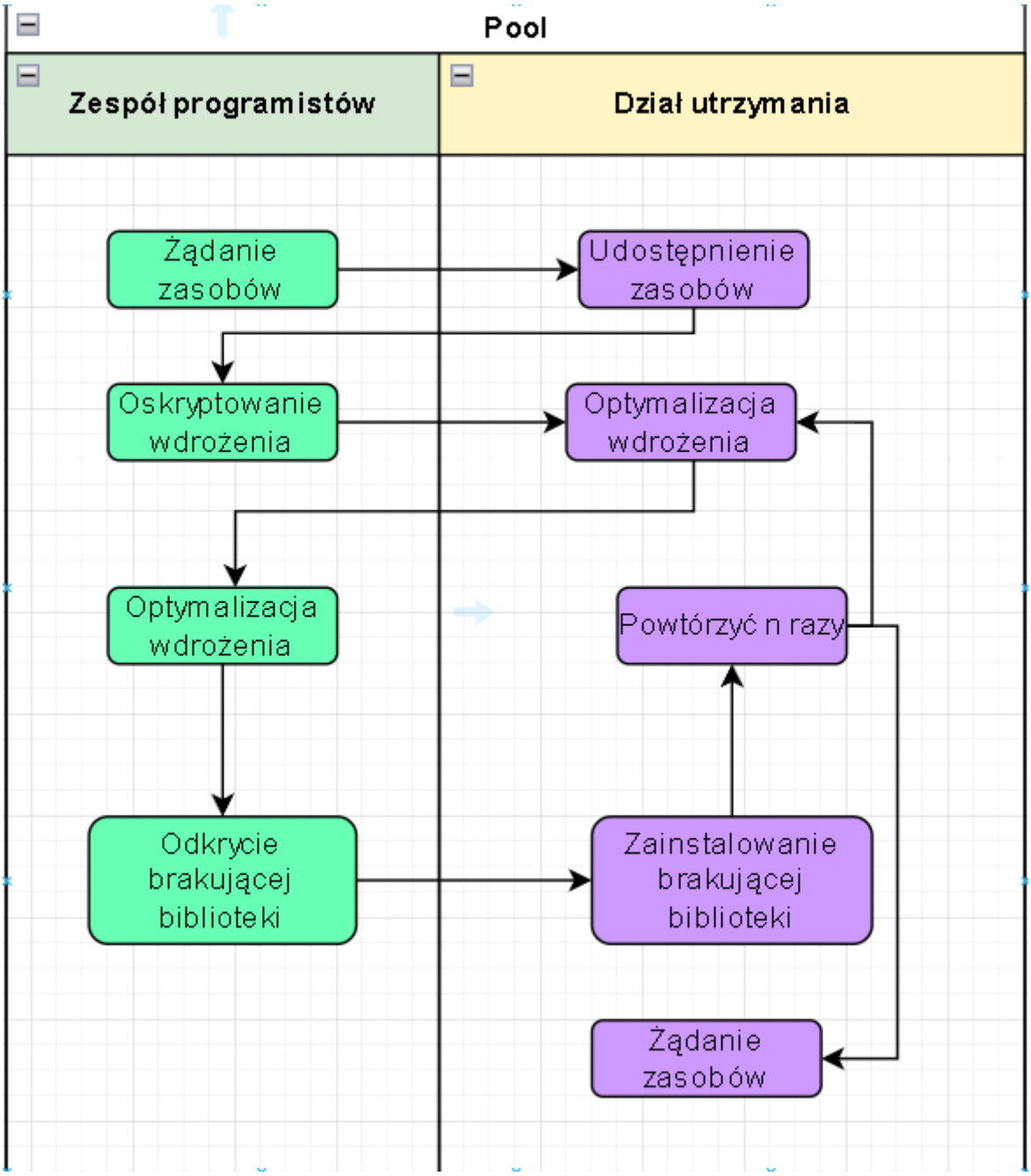
Zalety kontenerów Dockera

Modularność aplikacji osiągana za pomocą wirtualizacji i konteneryzacji jest normą zwinności w IT. Czynniki, które przyczyniły się do spopularyzowania konteneryzacji to:

1. Wydajność – kontenery współdzielą zasoby, rozkładanie obciążeń roboczych pozwalał uzyskać wyższą wydajność od rozwiązań jednozadaniowych. Różne zasoby mogą być obwarowywane ograniczeniami i przekazywane bezpośrednio aplikacjom. Liczba kontenerów, które można uruchomić na hoście jest

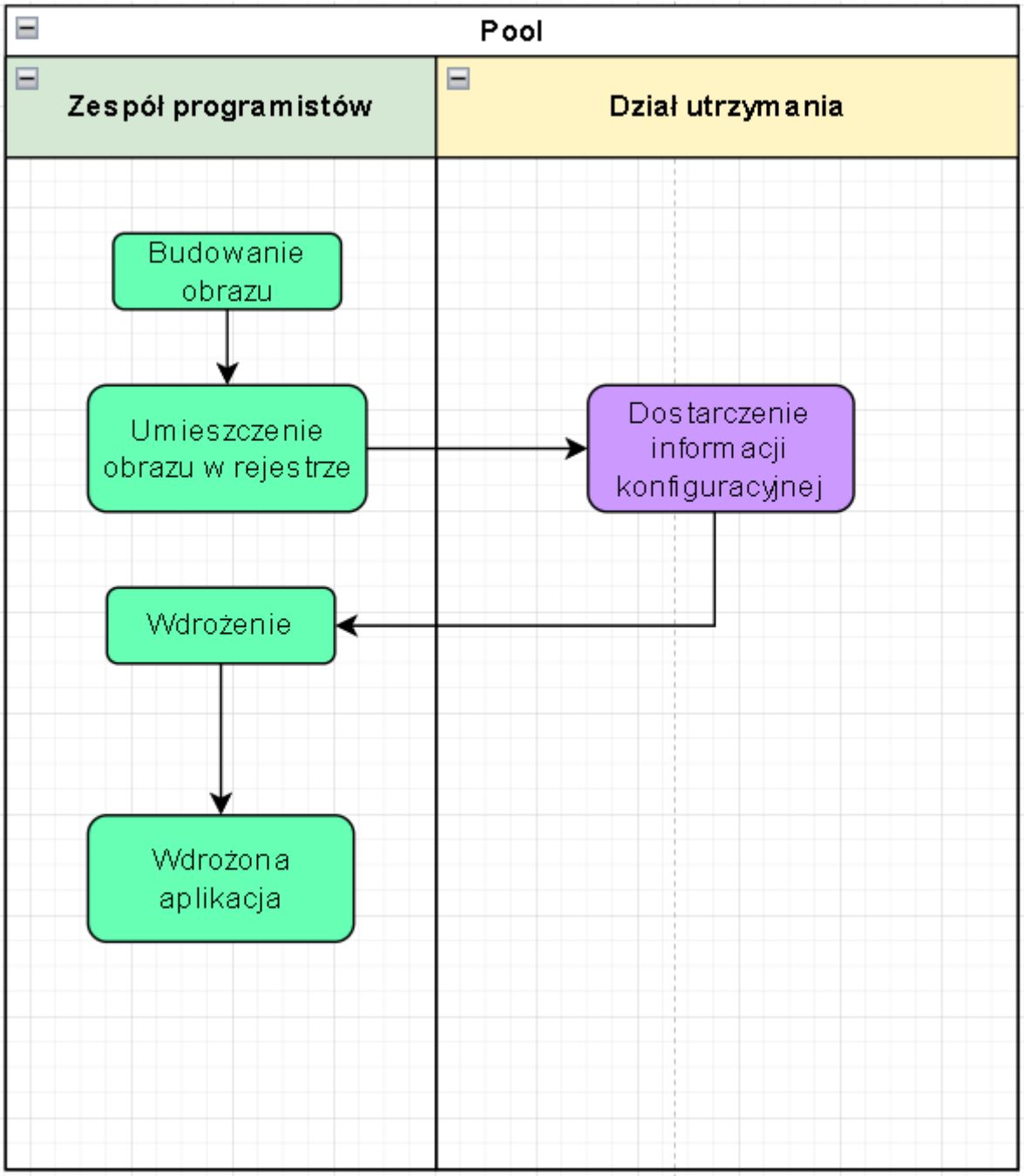
porównywalna z liczba maszyn wirtualnych, które można uruchomić na fizycznym sprzęcie.

2. Przenośność – środowisko uruchomieniowe aplikacji jest umieszczone we wspólnym systemie plików. Aplikacje mogą zostać uruchomione w dowolnym środowisku (lokalnie, w maszynie wirtualnej, na fizycznym serwerze)
3. Skalowalność – dowolna liczba nowych kontenerów może w ciągu sekund zostać obwarowana ograniczeniami umożliwiającymi obsługę obciążeń danych. Działanie kontenerów może zostać przerwane, gdy przestaną być potrzebne. Umożliwia to zwiększenie przepustowości i uruchamianie na życzenie.
4. Dostępność – w przypadku przerwania pracy jednego kontenera, usługa jest świadczona przez pozostałe kontenery. Dzięki orkiestracji mogą być automatycznie przywracane.
5. Zwrotność – aplikacje uruchamiane w kontenerach mogą być aktualizowane, modyfikowane bez wpływu na inne kontenery.
6. Tworzenie grup – kontenery w razie konieczności mogą zostać zagregowane.
7. Przewidywalność – obrazy dzięki niezmiennej naturze zawsze działają tak samo.
8. Pełna kontrola kodu – programiści mają pełną kontrolę nad kodem
9. Skierowanie na mikrousługi – kontenery doskonale nadają się do umieszczania w nich mikrousług
10. Wpieranie metodyk zwinnych – konteneryzacja sprawia, że zwinne metodyki DevOps stają się wszechobecne oraz maksymalnie przyśpiesza uruchamianie systemu. Ten sam obraz może być użyty przez środowiska deweloperskie, testowe, produkcyjne. Agitacja przeprowadzona na poniższych rysunkach.



Rys. 6. Przebieg tradycyjnego wdrożenia bez Dockera

Źródło: Opracowanie własne na podstawie [2]



Rys. 7. Przebieg wdrożenia z Dockerem

Źródło: Opracowanie własne na podstawie [2]

Porównanie Podatności i Krytyczności w Środowiskach Kontenerowych, Maszynach Wirtualnych i Wirtualizacji Enterprise

1. Konteneryzacja

W ostatnich latach zgłoszono szereg krytycznych podatności dotyczących silników kontenerowych (np. Docker, runc). Pomimo stosunkowo częstych aktualizacji, ataki oparte na eskalacji uprawnień czy „container escape” pozostają realnym zagrożeniem.

Do analizy obrazów kontenerowych warto wykorzystać narzędzia takie jak Clair, Anchore Engine czy Trivy, które automatycznie identyfikują znane luki w oprogramowaniu.

2. Maszyny Wirtualne (VMware, VirtualBox)

Choć podatności w hypervisorach (np. w VMware lub VirtualBox) są wykrywane, ich liczba jest stosunkowo niewielka ze względu na rygorystyczne procedury aktualizacji i weryfikację kodu. Jednakże krytyczność wykrytych luk bywa bardzo wysoka, ponieważ atak na hypervisor może narazić cały system. Skanery bezpieczeństwa, takie jak Nessus czy OpenSCAP, mogą być używane do oceny podatności zarówno hosta, jak i hypervisorów, co umożliwia monitorowanie stanu zabezpieczeń maszyn wirtualnych.

3. Wirtualizacja Enterprise

Choć żaden system nie jest odporny na ataki, środowiska enterprise charakteryzują się mniejszą liczbą zgłaszanych podatności oraz szybszą reakcją na zagrożenia. W praktyce krytyczność wykrytych luk jest starannie oceniana i korygowana przez producentów. Wdrożenia enterprise często korzystają z zaawansowanych narzędzi do monitorowania bezpieczeństwa, takich jak Splunk, ELK Stack czy systemy SIEM, które zbierają logi i umożliwiają analizę incydentów w czasie rzeczywistym.

4. Podsumowanie i Wnioski

Dla Kontenerów częściej występujące luki wynikające z dzielenia wspólnego jądra; krytyczne zagrożenia typu „container escape” wymagają ciągłych aktualizacji i odpowiedniej konfiguracji zabezpieczeń. Podatności dla maszyn wirtualnych są mniej liczne, ale często krytycznych podatności w hypervisorach; pełna izolacja systemowa zwiększa bezpieczeństwo, jednak atak na hypervisor może mieć katastrofalne skutki. Najmniejsza liczba podatności występuje w Enterprise Virtualization. Dzięki wysokim standardom bezpieczeństwa i systematycznym audytom; złożoność systemu wymaga zaawansowanego zarządzania, ale inwestycje w bezpieczeństwo minimalizują ryzyko. Każde z rozwiązań ma swoje mocne i słabe strony – kontenery oferują lekkość i szybkość, ale wymagają szczególnej uwagi przy konfiguracji zabezpieczeń; maszyny wirtualne gwarantują solidną izolację, lecz ich bezpieczeństwo zależy od odporności hypervisora; systemy enterprise inwestują w najwyższe standardy zabezpieczeń, co minimalizuje ryzyko, choć nie eliminuje go całkowicie.

Przygotowanie środowiska:

W ramach mojego projektu wykorzystuję Docker Desktop działający w środowisku WSL na systemie Windows. Konteneryzacji podlega tutaj jedynie frontend aplikacji napisanej w React. Proces budowania aplikacji opiera się na obrazie Node.js (node:18), który służy

do instalacji zależności oraz wygenerowania produkcyjnej wersji aplikacji. W fazie uruchomieniowej (runtime) statyczne pliki Reacta serwowane są za pomocą lekkiego i wydajnego serwera Nginx opartego na obrazie nginx:stable-alpine. Równolegle, na maszynie wirtualnej z systemem Ubuntu 20.04.4 uruchomionej w VirtualBox, skonfigurowałem osobny serwer Nginx, który serwuje dokładnie te same statyczne pliki Reacta, co kontener. Takie podejście pozwala na przeprowadzenie porównawczych testów wydajnościowych, zestawiając działanie aplikacji w kontenerze Dockerowym z jej funkcjonowaniem w tradycyjnym środowisku serwera na maszynie wirtualnej.

Konfiguracja sprzętowa środowiska testowego

Komponent	**Specyfikacja**
CPU	Intel Core i3-10100F, 4 rdzenie, 8 wątków, 3.60 GHz
GPU	Nvidia GTX 1050 Ti
RAM	32 GB DDR4 2666 MHz (dual channel)
Płyta główna	Gigabyte H410M S2H V3
System operacyjny	Windows 10 64-bit

Konfiguracja środowisk wirtualnych

Środowisko	**Parametry**
Maszyna wirtualna (VirtualBox)	Ubuntu 20.04.4 LTS
	20 GB RAM
	4 procesory
Docker Desktop (WSL 2)	20 GB RAM
	4 procesory

Przeprowadzenie doświadczeń:

Wszystkie testy obciążeniowe zostały przeprowadzone przy użyciu narzędzia Apache Benchmark (ab). Każde polecenie uruchomiono 100-krotnie, a wyniki zapisywano do osobnych plików tekstowych.

Test	-n (żądania)	-c (równoległość)	URL
1. Szybki test responsywności	100	10	/
2. Średnie obciążenie (standardowy)	1000	50	/
3. Stres-test (duże obciążenie)	5000	200	/
4. Test opóźnień (niska równoległość)	500	1	/

Aby porównać wydajność kontenera Docker oraz maszyny wirtualnej, wykonano pojedynczy, długotrwały test: ***ab -n 2000000 -c 500 http://<ip_hosta>/***

Monitorowanie zasobów kontenera w czasie testu prowadzono za pomocą

```
docker stats $ContainerId --no-stream --format
"{{.Container}},{{.Name}},{{.CPUPerc}},{{.MemUsage}},{{.MemPerc}},{{
.NetIO}},{{.BlockIO}},{{.PIDs}}"
```

Monitorowanie maszyny wirtualnej (Ubuntu w VirtualBox) przeprowadzono przy pomocy `top` w trybie batch: `top -b -d 2 -n 600 -p "$PIDS" > nginx_top.log`

Dzięki tak przygotowanym danym można porównać zarówno czasy odpowiedzi i przepustowość serwera, jak i zużycie zasobów w środowisku Docker vs. maszynie wirtualnej.

Analiza wyników i wnioski:

Porównanie testu „vbox” vs „kontener” ab -n 2000000 -c 500

Tabela 1 wyniki wydajności virtualbox vs kontener

Metryka	Kontener Docker	VM (VirtualBox)	Uwagi
Requests/sec	2 368,7 [/s]	8 069,5 [/s]	VM ~3.4× wyżej
Time per request (mean)	211,08 ms	61,96 ms	niższa latencja na VM
Time per request (all concurrent)	0,422 ms	0,124 ms	-
Transfer rate	1 600,8 KB/s	5 524,1 KB/s	VM ~3.5× wyżej

Connect (mean)	0 ms	30 ms	-
Processing (mean)	211 ms	32 ms	VM szybciej obsługuje request
Waiting (mean)	145 ms	22 ms	czas oczekiwania na odpowiedź
95 percentyl Total	261 ms	79 ms	VM znacznie węższy rozkład latencji

1. Wydajność throughput
   – VM obsłużyła ponad 8 000 żądań/s, kontener jedynie ~2 400 żądań/s. Różnica (~3,4×) wynika głównie z narzutu warstwy WSL2/Docker Desktop oraz możliwych różnic w konfiguracji sieciowej.
2. Latencja
   – Kontener cechuje się średnią obsługą żądania na poziomie 211 ms, podczas gdy VM — 62 ms. Również 95 percentyl latencji to 261 ms vs 79 ms.
3. Transfer rate
   – Przepustowość VM (5 524 KB/s) jest prawie 3,5× wyższa niż w kontenerze (1 600 KB/s), co wskazuje na lepszą wydajność I/O w bezpośredniej maszynie wirtualnej.
4. Czasy sieciowe
   – W kontenerze łączność (Connect) praktycznie nie generuje opóźnień (0 ms), bo używa host network. VM wypada gorzej (średnio 30 ms), ale rekompensuje to szybsze przetwarzanie.

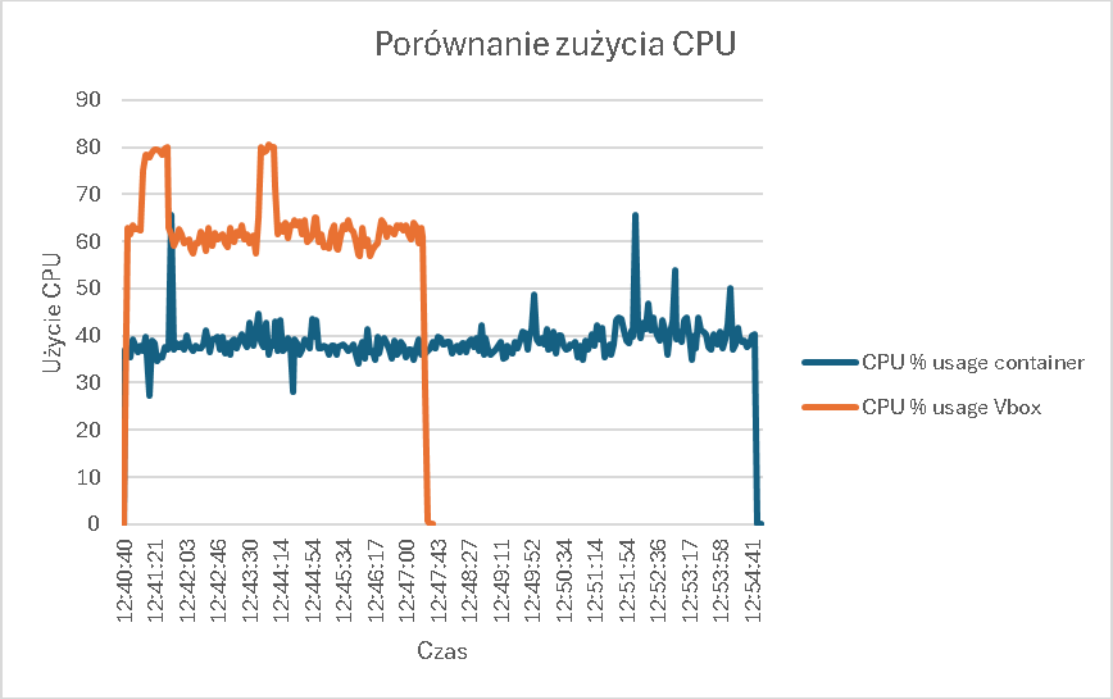
Konteneryzacja w środowisku Docker Desktop + WSL2 w tej konfiguracji generuje istotny narzut wydajnościowy — zarówno pod względem przepustowości, jak i latencji. Maszyna wirtualna z Ubuntu na VirtualBox oferuje znacznie lepsze osiągi dla serwera WWW w testach wysokonakładowych. Przy projektowaniu środowisk produkcyjnych warto:

1. Testować w warunkach możliwie zbliżonych do docelowych (kontenery vs bare-metal vs VM).
2. Rozważyć użycie host networking lub bezpośredniego uruchomienia Nginx na hoście dla krytycznych usług.
3. Zmieniać ustawienia Docker/WSL (limit CPU, I/O, bardziej wydajne sterowniki).

Poniższe wykresy pokazują, jak kontener i maszyna wirtualna (VBox) zachowywały się podczas testu obciążeniowego Apache Bench:

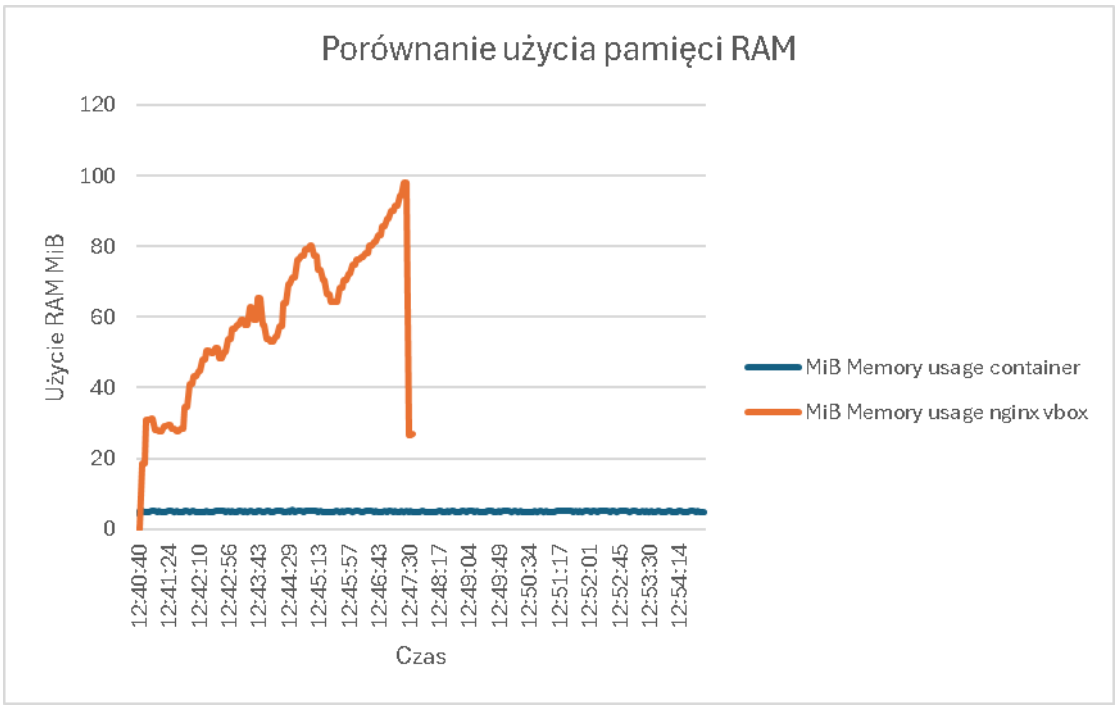
1. CPU
    a. Średnie obciążenie kontenera: ~37,6 % (od 0 % do 65,5 %)
    b. Średnie obciążenie VM: ~62,6 % (od 0 % do 80 %)
    c. VM przyjęła większy udział CPU, a kontener utrzymywał stabilne ~35–40 % przez większość testu, z jednym skokiem ~65 %.
2. Pamięć
    a. Kontener zużywał stabilnie ok. 5 MiB RAM.
    b. RAM serwera nginx uruchomionego pod VBox urósł od ~0 do ~63 MiB w trakcie testu i osiągnął średnio ~41 MiB.

Test generuje umiarkowane obciążenie CPU w kontenerze, ale VM w tle zużywa zdecydowanie więcej mocy obliczeniowej. Zużycie pamięci kontenera jest marginalne w porównaniu do VM — znaczna część narzutu pamięci generuje sama maszyna wirtualna.



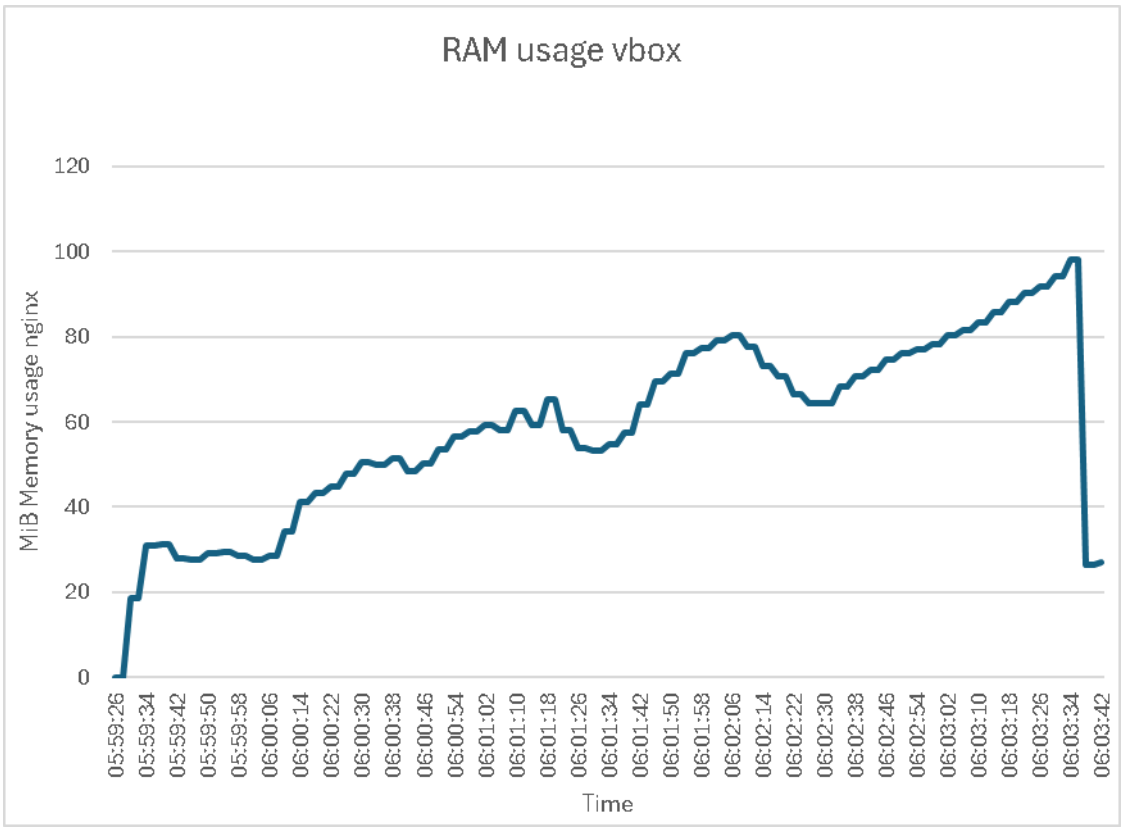
Rys. 8. Porównanie zużycia CPU między kontenerem a maszyną wirtualną

Źródło: Opracowanie własne



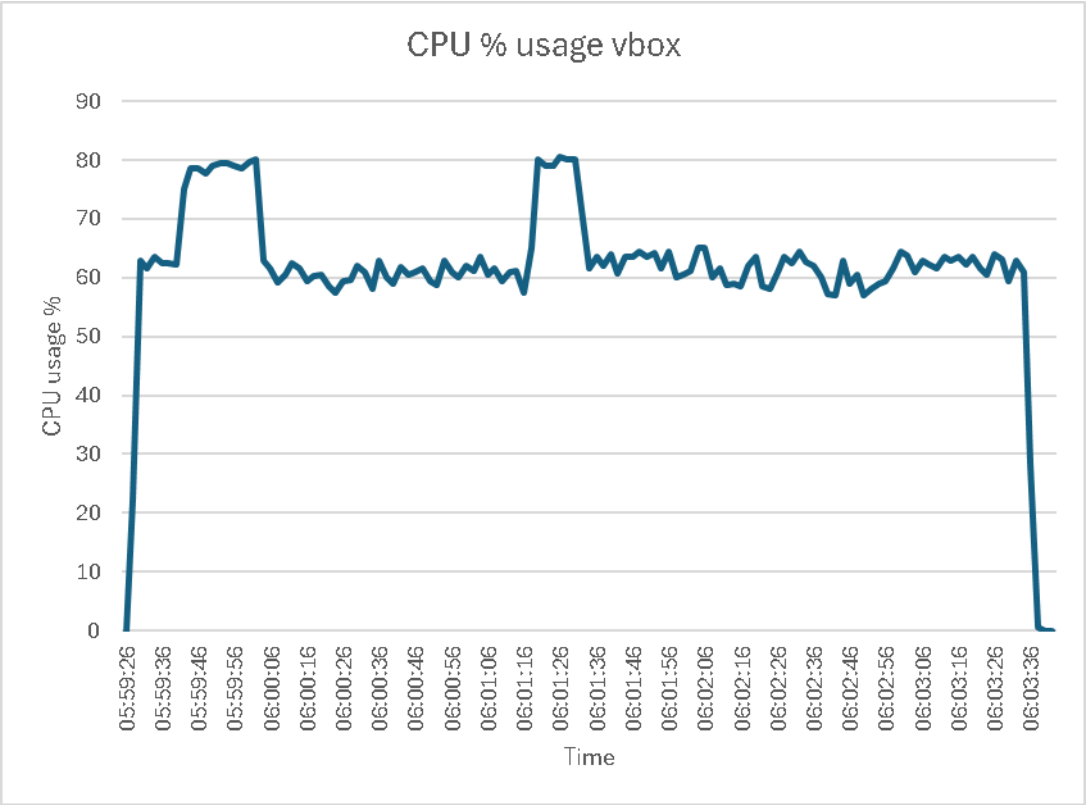
Rys. 9. Porównanie zużycia pamięci RAM między kontenerem a maszyną wirtualną

Źródło: Opracowanie własne



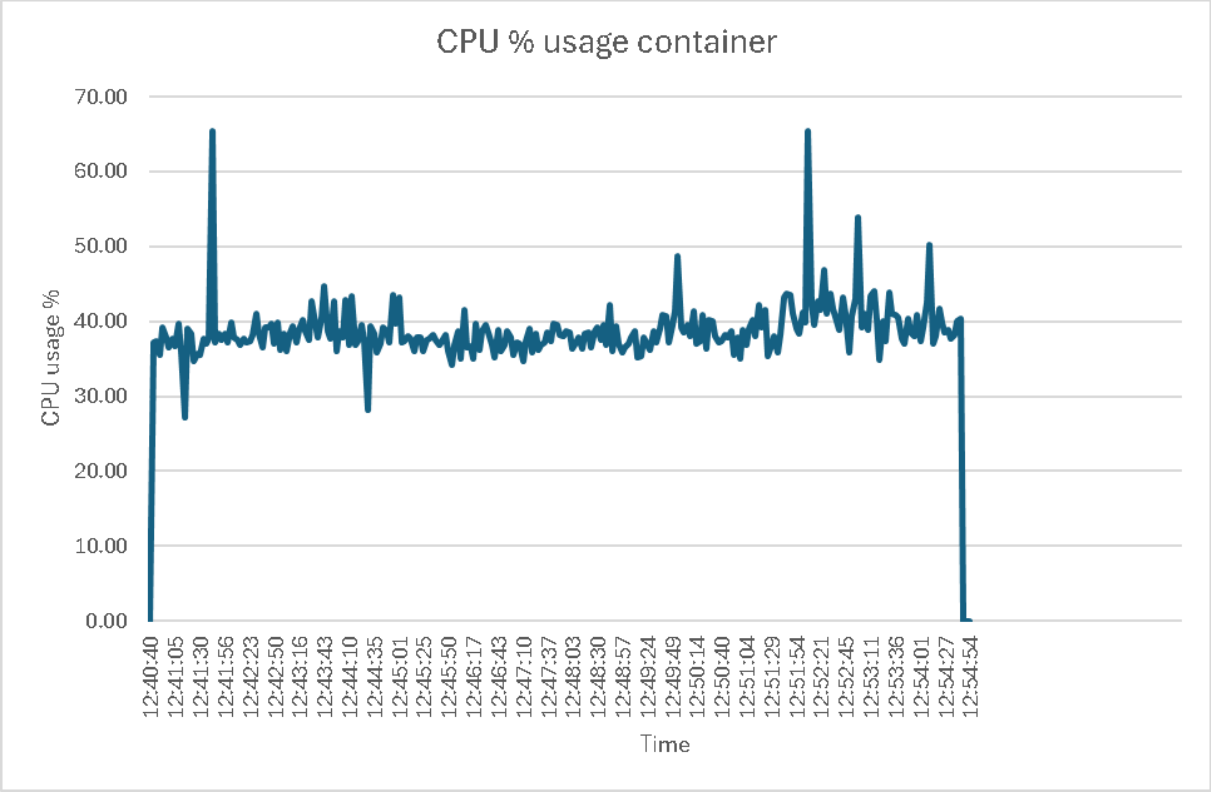
Rys. 10. Zużycie pamięci RAM przez maszynę wirtualną

Źródło: Opracowanie własne



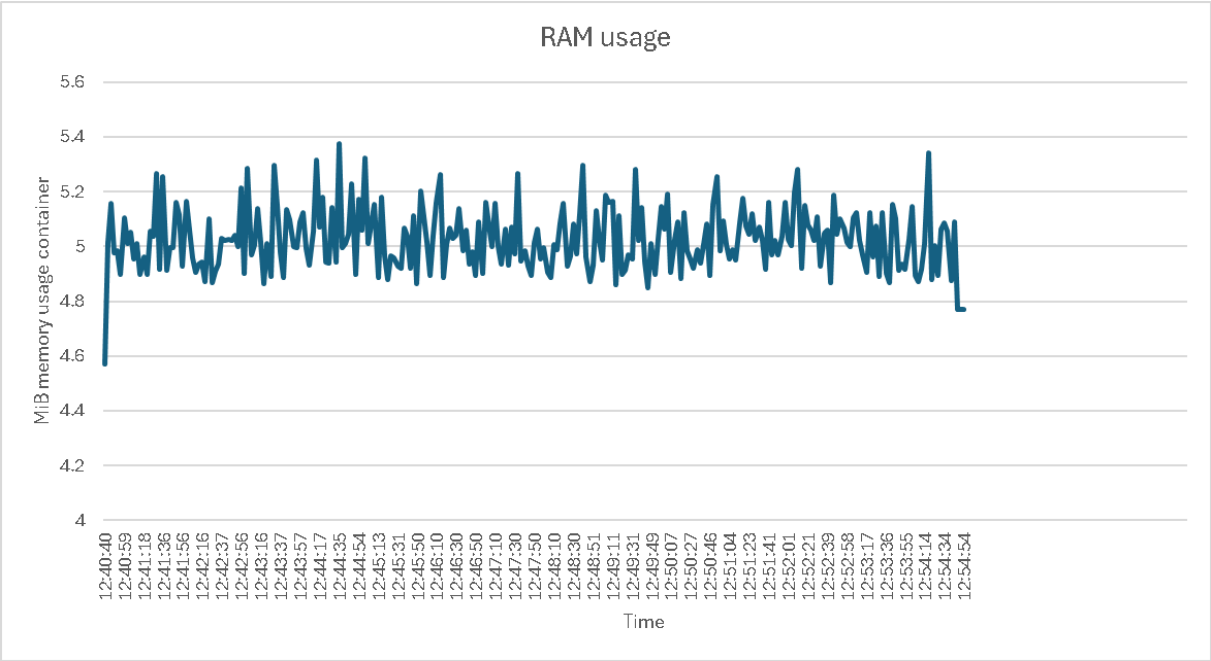
Rys. 11. Zużycie CPU przez maszynę wirtualną

Źródło: Opracowanie własne



Rys. 12. Zużycie CPU przez kontener

Źródło: Opracowanie własne



Rys. 13. Zużycie pamięci RAM przez kontener

Źródło: Opracowanie własne

Porównanie testu „vbox" vs „kontener" ab -n 5000 -c 200

Poniżej zestawienie kluczowych metryk z kolejnych uruchomień ApacheBench dla:

- VM (VirtualBox, Nginx 1.18.0) na porcie 80
- Kontenera Docker (nginx:stable-alpine 1.26.2) na porcie 9090

Tabela 2 Requests per second (średnia i rozrzut)

Środowisko	Min	Max	Średnia	Odchylenie std.	Zakres w %
VM (vbox)	6 164/s	8 874/s	7 625/s	≈ 900/s (12 %)	–19 % … +16 %
Kontener	2 124/s	2 372/s	2 299/s	≈ 95/s (4 %)	–8 % … +3 %

- VM jest ok. 3,3× szybsza niż kontener pod względem przepustowości.
- Testy na VM wykazują większą zmienność (± 19 %); kontener jest bardziej stabilny (± ≈ 5 %).

Tabela 3 Czas na żądanie (mean Time per request)

Środowisko	Średnia całkowita [ms]	±std [ms]
VM (vbox)	25,0	3,7
Kontener	85,0	6,0

- Średnia latencja (czas – total) w VM to ~25 ms, w kontenerze ~85 ms.
- Kontener generuje ok. 3,4× wyższą latencję.

Tabela 4 Transfer rate

Środowisko	Średnia [KB/s]	Zakres
VM (vbox)	5 450	4 220–6 075
Kontener	1 580	1 390–1 604

- VM osiąga ok. 5 450 KB/s, kontener 1 580 KB/s (≈ 3,4× różnicy).

Tabela 5 Rozkład latencji (median vs 95 percentyl)

Środowisko	median Total [ms]	95 percentyl [ms]
VM (vbox)	24 ms	26 ms
Kontener	84 ms	93 ms

- VM: większość (95 %) żądań obsłużona w ≤ 26 ms.
- Kontener: 95 % żądań w ≤ 93 ms — wyraźnie szerszy rozrzut.

VM w VirtualBox (Ubuntu + Nginx) znacznie przewyższa kontener Docker w WSL2 pod względem wydajności sieci i czasu odpowiedzi. Kontener wykazuje mniejszą zmienność wyników (mniejszy współczynnik zmienności), co może być zaletą w środowiskach produkcyjnych. VM ma większy rozrzut, ale również wyższe szczytowe osiągi.

Jeśli krytyczna jest maksymalna przepustowość i minimalna latencja warto użyć VM lub bare-metal. Gdy zależy na elastyczności konteneryzacji i przewidywalności Docker w WSL2 może być akceptowalny, ale warto stroić limity zasobów i sieć.

Porównanie testu „vbox” vs „kontener” ab -n 100 -c 10

1. Bardzo wysoka przepustowość / niska latencja
    a. Oba środowiska – zarówno VM (VirtualBox) jak i kontener – radzą sobie z małym obciążeniem praktycznie bez wąskiego gardła.
    b. Średnia liczba żądań na sekundę (Requests/sec) utrzymuje się w przedziale 5 000–10 000 RPS, co dla 100 żądań przy 10 równoległych połączeniach oznacza pojedyncze milisekundy opóźnienia.
2. Średni czas obsługi (Time per request)
    a. Średnia wartość `Time per request` wynosi około 1–2 ms.
    b. Mediana (50 %) trafia w 1–2 ms, a 95 percentyl w 2–3 ms — niemal wszystkie odpowiedzi mieszczą się w tym przedziale.
3. Transfer rate
    a. Przepustowość mierzonego transferu oscyluje między 3 500 a 7 000 KB/s, adekwatnie do chwilowego obciążenia i wahań chwilowych.

4. Stabilność wyników
    a. Przy tak niskim wolumenie (100 żądań) wyniki bywają niekiedy „nieregularne” (ostrzeżenia o niestabilnym rozkładzie czasu oczekiwania) — to efekt bardzo krótkich testów i układów cache, a nie praktyczne ograniczenie serwera.
    b. Różnice między kolejnymi przebiegami są niewielkie i mieszczą się w granicach odchylenia rzędu 0,5–1 ms.
5. Wpływ środowiska
    a. VM może osiągać odrobinę wyższe piki RPS (np. ~9 000–10 000), ale też większą rozbieżność między kolejnymi przebiegami.
    b. Kontener (Docker Desktop + WSL2) w tym scenariuszu generuje niemal identyczne czasy obsługi — przy tak niskim obciążeniu narzut konteneryzacji jest praktycznie niezauważalny.

Dla lekkich, rzadkich żądań (niska równoległość) obie platformy są równoważne pod kątem responsywności. Przy małym obciążeniu kontener nie wprowadza istotnych opóźnień — możesz swobodnie korzystać z Docker Desktop bez utraty wydajności. Ostrzeżenia o „nierealistycznym” rozkładzie czasu w ApacheBench sugerują, by w podobnych scenariuszach wydłużyć test (więcej żądań), by uzyskać stabilniejsze statystyki.

Porównanie testu „vbox” vs „kontener” ab -n 500 -c 1

1. Przepustowość
    a. Na VM Nginx potrafił obsłużyć średnio ponad 5 000 RPS, podczas gdy w kontenerze jedynie ~ 750 RPS – to ok. 7× spadek przepustowości.
2. Latencja
    a. VM: 0,1 ms średniego czasu obsługi żądania (mediana 0 ms!).
    b. Kontener: 1,3 ms średniego czasu (mediana 1 ms).
    c. Czyli kontener wprowadza dodatkowe ~ 1 ms opóźnienia.
3. Transfer danych
    a. VM: ~ 3 700 KB/s, kontener: ~ 500 KB/s – również około 7× wolniej.
4. Stabilność i błędy
    a. Zarówno na VM jak i w kontenerze: 0 błędów.
    b. Na VM większa zmienność wyników (2 400–8 900 RPS) wskazuje na wpływ stanu cache i I/O hypervisora.

W tym scenariuszu (pojedyncze, sekwencyjne żądania) – mimo że konteneryzacja ułatwia izolację i deployment – daje ona wyraźny narzut na wydajność w porównaniu do “czystej” VM: ok. 7–8× niższa przepustowość i 10× wyższa latencja per request.

Porównanie testu „vbox” vs „kontener” ab -n 100 -c 50

1. Przepustowość (RPS)
    a. Średnio VM osiągała ~8 700 req/s, kontener tylko ~2 300 req/s – różnica ~4×.
2. Opóźnienia (czas całkowity)
    a. Średnie czasy w VM to ~6 ms, podczas gdy w kontenerze ~21 ms (4× wolniej).
3. Stabilność wyników
    a. VM ma nieco większe rozstęp między testami (6 979–11 689), ale i tak zawsze znacznie szybciej niż kontener.
    b. Kontener pokazuje konsekwentnie niższą, choć bardziej zbliżoną do siebie, przepustowość.
4. Przyczyny różnic
    a. Dodatkowe narzuty sieci/konteneryzacji (CNI, mosty sieciowe, namespace’y), inne ustawienia jądra/IO.

Jeśli kluczowa jest maksymalna wydajność w prostych, stateless’owych serwerach HTTP, VM może dać wyraźnie lepsze rezultaty. Kontenery za to upraszczają deployment i izolację – tu trzeba balansować między wygodą a potrzebą surowej przepustowości.

Wnioski z tesów „VBox” vs „Kontener” na różnych scenariuszach Apache

1. Znacznie wyższa przepustowość VM
    a. We wszystkich testach maszyna wirtualna (Ubuntu + VirtualBox) osiągała 3–8× wyższą liczbę żądań na sekundę niż ten sam serwer uruchomiony w kontenerze Docker Desktop + WSL2.
    b. Przykładowo przy `-n 2000000 -c 500` VM osiągnęła ~8 069 rps vs ~2 369 rps (ok. 3,4× przewagi).
2. Niższa latencja na VM
    a. Średni czas obsługi żądania („Time per request”) w VM był 3–10× mniejszy niż w kontenerze, w zależności od obciążenia:
        i. Przy dużym obciążeniu (`-n 2 000 000 -c 500`): 62 ms vs 211 ms.
        ii. Przy sekwencyjnym połączeniu (`-n 500 -c 1`): 0,15 ms vs 1,37 ms.
    b. Również 95 percentyle latencji w VM pozostawały znacznie niższe i bardziej zwarte (np. 79 ms vs 261 ms przy `-n 2 000 000 -c 500`).
3. Narzut konteneryzacji i WSL2

    a. Konteneryzacja w Docker Desktop na WSL2 wprowadza istotne narzuty sieciowe, I/O i kontekstowe, które rosną wraz z poziomem równoległości i wolumenem żądań.
    b. Przy bardzo małej liczbie równoległych połączeń (`-n 100 -c 10`) różnica praktycznie zanika (1–2 ms vs 1–2 ms), ale już przy rosnącym obciążeniu staje się wyraźna.
4. Stabilność wyników
    a. VM zwykle prezentuje większy rozrzut między kolejnymi uruchomieniami (szczególnie przy niskich i średnich obciążeniach), ale też może osiągać wyższe maksima.
    b. Kontener pokazuje bardzo przewidywalne wyniki (niższa zmienność latencji), co może być zaletą w środowisku produkcyjnym, jeżeli akceptowalny jest niższy poziom wydajności.
5. Rekomendacje
    a. Kiedy priorytetem jest maksymalna przepustowość i najniższa latencja lepszym wyborem jest VM (lub bare-metal).
    b. Kiedy kluczowa jest elastyczność, izolacja i przewidywalność konteneryzacja może być dobrym kompromisem, pod warunkiem optymalizacji.

Podsumowanie

Celem opracowania było przedstawienie koncepcji konteneryzacji z wykorzystaniem Dockera – od omówienia jego architektury i mechanizmów izolacji, przez kwestie bezpieczeństwa i zarządzania zasobami, aż po monitorowanie działania – a także praktyczna weryfikacja wpływu konteneryzacji na wydajność serwera WWW w porównaniu z maszyną wirtualną.

W części teoretycznej opisano podstawowe mechanizmy izolacji w Dockerze oparte na namespaces i cgroups. W dziale poświęconym bezpieczeństwu omówiono wyzwania wynikające ze współdzielenia jądra systemu gospodarza, wskazując na możliwości ograniczania ryzyka za pomocą profilu SELinux. Kolejna część pracy poświęcona została praktycznym aspektom zarządzania danymi, siecią i logami. Zaprezentowano podejście do gromadzenia logów poprzez różne driver'y (JSON-file, syslog) i mechanizmy monitorowania metryk kontenerów przy użyciu narzędzi takich jak Prometheus, co umożliwia wczesne wykrywanie wąskich gardeł. W części eksperymentalnej przygotowano dwa środowiska: Ubuntu z Nginx uruchomione w VirtualBox oraz identyczną aplikację w kontenerze Docker Desktop działającym na WSL2. Do pomiarów wydajności wykorzystano ApacheBench w pięciu scenariuszach: od niewielkiego obciążenia (100 żądań z 10 równoległymi połączeniami) aż po intensywny test dwóch milionów żądań przy 500 równoczesnych połączeniach. Mierzono liczbę żądań na

sekundę, średnie i percentylowe czasy odpowiedzi oraz przepustowość transferu danych. Wyniki jednoznacznie pokazały przewagę maszyny wirtualnej w testach o wysokim i średnim natężeniu ruchu. Podczas testu dwóch milionów żądań VM osiągnęła ponad osiem tysięcy żądań na sekundę przy średniej latencji około 62 ms, natomiast kontener zaledwie około 2 400 rps i latencji 211 ms. Podobny stosunek ok. 3–4× różnicy obserwowano w testach 5 000 żądań przy 200 połączeniach oraz 1 000 żądań przy 50 połączeniach. Przy niewielkim obciążeniu (100 rps, 10 wątków) oba środowiska radziły sobie niemal identycznie, a różnice w opóźnieniach rzędu 1–2 ms były pomijalne. Z kolei w sekwencyjnym teście 500 żądań z jednym połączeniem kontener wykazywał nawet siedmiokrotnie wyższe czasy obsługi niż VM.

Na podstawie przeprowadzonych badań można wyciągnąć następujące wnioski: konteneryzacja w Dockerze jest doskonałym narzędziem tam, gdzie priorytetem są szybkość wdrożeń, spójność środowisk i elastyczne skalowanie mikroserwisów, natomiast w scenariuszach wymagających ekstremalnej przepustowości i niskiej latencji – zwłaszcza przy bardzo dużym ruchu – lepiej sprawdzi się uruchomienie aplikacji na maszynie wirtualnej albo bezpośrednio na bare-metal. Aby złagodzić narzut Docker Desktop/WSL2, warto rozważyć optymalizację limitów CPU i I/O kontenerów oraz stosowanie lekkich obrazów z minimalną liczbą warstw.

Bibliografia

[1] https://docs.docker.com
[2] Docker dla praktyków, Helion 2018. Jeeva S. Chelladhurai, Vinod Singh, Pethuru Raj
[3] Docker. Niezawodne kontenery produkcyjne. Helion 2024. Sean P.Kane, Karl Matthias