Kontenery aplikacyjne

mgr inż. Jakub Woźniak

Zarządzanie Systemami Rozproszonymi
Instytut Informatyki
Politechnika Poznańska

Plan wykładu

- Wprowadzenie do wirtualizacji i rys historyczny
- Potrzeba i charakterystyka wirtualizacji
- Konteneryzacja jako odpowiedź na wady wirtualizacji
- Przykłady technologii konteneryzacji
- Nawiązanie do problemów w logistyce
- Docker jako kontener aplikacyjny

Wprowadzenie

Wirtualizacja: od IBM CP-40 do współczesnych hypervisorów

Konteneryzacja jako lżejsza forma wirtualizacji

Docker i DevOps – kluczowe elementy nowoczesnych wdrożeń aplikacji

Wirtualizacja – rys historyczny

- **1967 – IBM CP-40**: Początek wirtualizacji na potrzeby optymalizacji zasobów mainframe.
- **1998 – VMware**: Wprowadzenie komercyjnego hypervisora VMware, który umożliwia wirtualizację na serwerach x86.
- **2008 – Microsoft Hyper-V**: Microsoft dołącza do gry z Hyper-V, wirtualizatorem natywnie wbudowanym w system Windows Server.
- **2010+ – Rozwój KVM (Kernel-based Virtual Machine)**: Open-source'owa wirtualizacja w jądrze Linuksa, szeroko używana w chmurach obliczeniowych.

Potrzeba wirtualizacji

- **Optymalizacja zasobów**: Możliwość uruchamiania wielu systemów operacyjnych na jednym serwerze
- **Izolacja**: Stabilność i bezpieczeństwo systemu
- **Elastyczność**: Uruchamianie różnych OS na jednym sprzęcie
- **Skalowalność**: Szybkie tworzenie nowych VM

Wirtualizacja adresuje problem fragmentacji, nie do końca go rozwiązując. Narzut wirtualizacji powoduje zwiększone zużycie zasobów w ogóle, mimo prawdopodobnego zmniejszenia liczby maszyn fizycznych.

Zalety i wady wirtualizacji

- **Zalety:**
  - Lepsze zarządzanie zasobami
  - Elastyczność
  - Bezpieczeństwo i izolacja
- **Wady:**
  - Narzut na wydajność (hiperwizor)
  - Złożoność zarządzania
  - Większe zasoby sprzętowe

Konteneryzacja – rozwiązanie problemów wirtualizacji?

- **Wirtualizacja:**
  - Każda maszyna wirtualna ma własne jądro systemu operacyjnego
  - Większe zużycie zasobów
  - Narzut na wydajność (hiperwizor)
- **Konteneryzacja:**
  - Kontenery współdzielą jądro systemu z gospodarzem
  - Lżejsze i szybsze uruchamianie aplikacji
  - Mniejsze zużycie zasobów

Przykłady tradycyjnych kontenerów

- **LXC (Linux Containers):** Konteneryzacja na poziomie systemu operacyjnego
- **OpenVZ:** Wczesny system konteneryzacji dla Linuksa
- **Porównanie z wirtualizacją:**
  - Mniejsze zużycie zasobów (brak pełnej wirtualizacji sprzętu)
  - Każdy kontener współdzieli jądro z gospodarzem

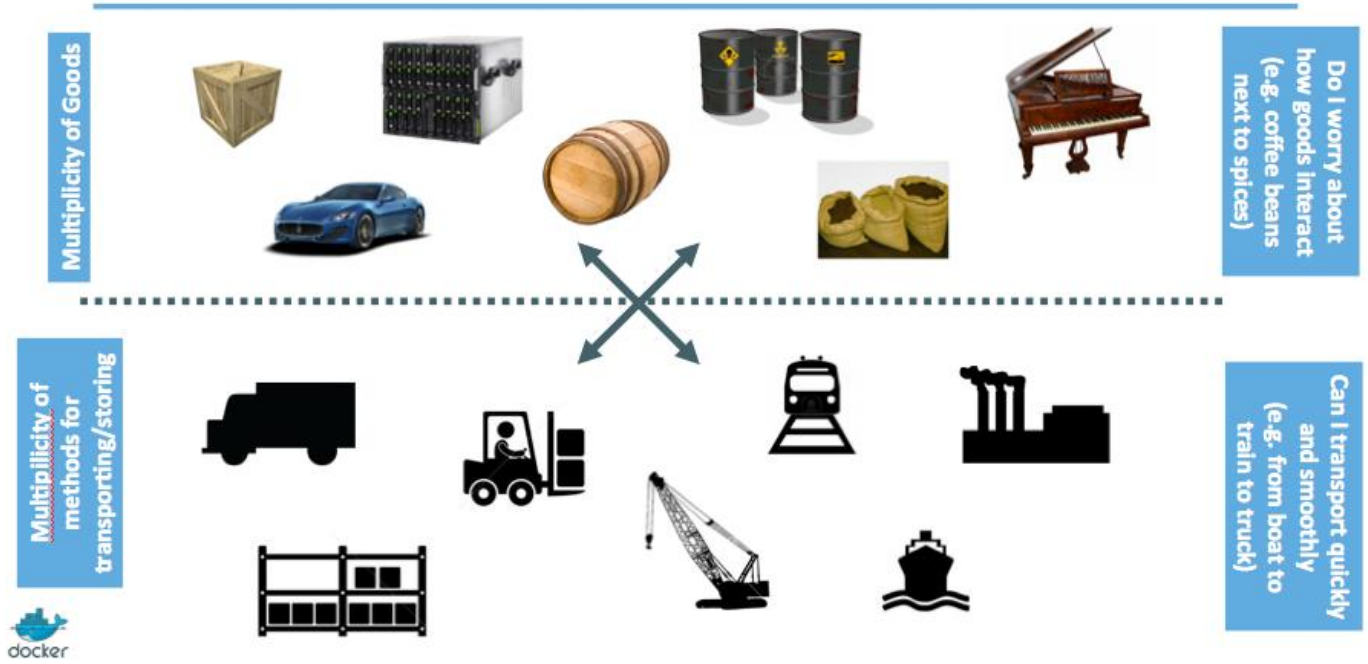
LXC i OpenVZ to dwa pierwsze systemy konteneryzacji, które umożliwiały izolację aplikacji bez potrzeby pełnej wirtualizacji sprzętu. Kontenery działają na poziomie systemu operacyjnego, co oznacza, że nie mają własnego jądra, ale współdzielą jądro z gospodarzem, co zmniejsza zużycie zasobów.

Kontener aplikacyjny

- **Kontener aplikacyjny:**
  - Lekka jednostka uruchomieniowa, która zawiera aplikację oraz jej zależności.
  - Główny proces (PID 1) to proces aplikacji (np. interpreter Python)
  - Zawiera minimalną warstwę systemu operacyjnego
- **LXC (Linux Containers):**
  - Kontener systemowy, uruchamia pełne środowisko systemu operacyjnego.
  - Główny proces to systemd lub init.
  - **Cel:** Izolacja i wirtualizacja na poziomie systemu operacyjnego.
  - **Przykład:** Wielozadaniowość na poziomie systemu.

Docker – kontener aplikacyjny w praktyce

- Docker ujednolica proces wdrażania aplikacji na różnych systemach
- Przenośność aplikacji (kontenery mogą działać na dowolnym systemie z Dockerem)
- Spójne środowisko adresujące problem „works on my computer”

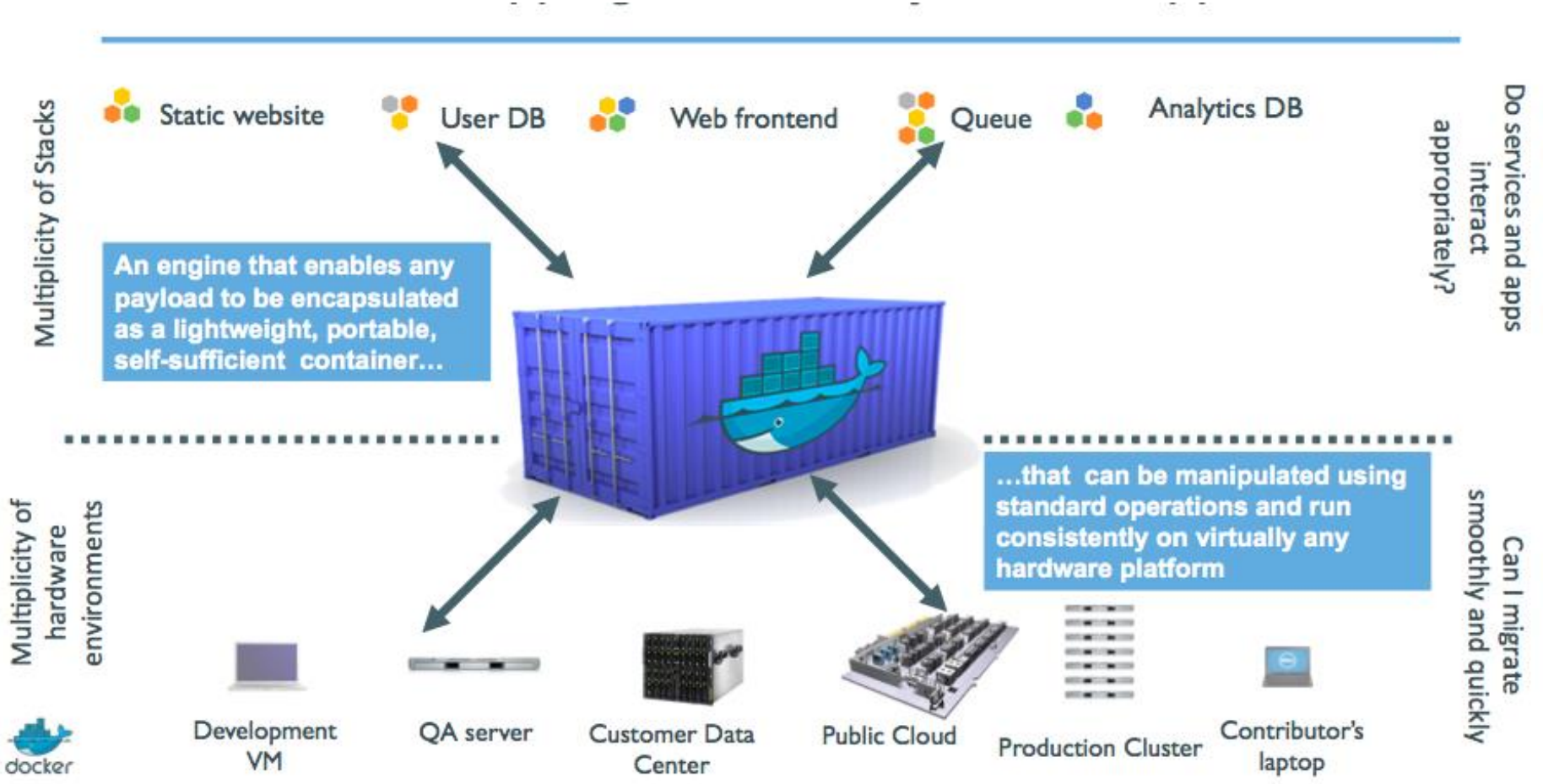


src: https://containers.goffinet.org/containers/011-docker_overview

Umieszczone w kontenerach

src: https://containers.goffinet.org/containers/011-docker_overview

Umieszczone w kontenerach (ale to IT)



src: https://containers.goffinet.org/containers/011-docker_overview

Docker – jak działa?

- **Warstwowy system plików (OverlayFS):**
  - Każdy obraz Docker składa się z wielu warstw.
  - Nowe warstwy są tworzone przy każdej zmianie w Dockerfile.
  - OverlayFS umożliwia ponowne wykorzystanie warstw, co przyspiesza budowę obrazów.
- **Kontrolne grupy zasobów (cgroups):**
  - Zarządzają zasobami kontenerów, np. CPU, pamięcią, I/O.
  - Zapewniają izolację zasobów między kontenerami.
  - Ograniczają zużycie zasobów przez poszczególne kontenery, chroniąc system gospodarza.
- **Podstawowe polecenia Docker CLI**:
  - docker build – buduje obraz
  - docker pull – pobiera obraz z rejestru
  - docker run – uruchamia kontener z podanego obrazu

Dockerfile

- **Struktura Dockerfile:**
  - **FROM:** Definiuje obraz bazowy
  - **RUN:** Wykonuje polecenia w czasie budowy obrazu
  - **COPY/ADD:** Kopiuje pliki do obrazu
  - **CMD/ENTRYPOINT:** Definiuje polecenie uruchamiane przy starcie kontenera
- **Optymalizacja:**
  - Używaj mniejszych obrazów bazowych (np. Alpine)
  - Minimalizuj liczbę warstw w Dockerfile
  - Multi-stage builds

Persistent Storage w Dockerze

- **Volumes:** Przechowywanie danych, zarządzane przez Docker
- **Bind Mounts:** Bezpośredni dostęp do katalogów na systemie gospodarza
- **Kiedy używać?**
  - **Volumes:** Gdy potrzebujesz niezależności od systemu gospodarza
  - **Bind Mounts:** Gdy aplikacja musi mieć dostęp do konkretnych plików na gospodarzu

Podman – alternatywa dla Docker

- **Podman**:
  - Narzędzie do zarządzania kontenerami bez procesu dockerd
  - **Zalety Podmana:**
    - Brak potrzeby uruchamiania demona
    - Zwiększone bezpieczeństwo (brak potrzeby uprawnień roota)
- **Kompatybilność z Dockerem:**
  - Możliwość używania tych samych poleceń, np. podman run

podman

Docker produkcyjnie

- **Logowanie i monitoring:**
  - Narzędzia takie jak ELK Stack, Prometheus, Grafana
- **Orkiestracja:**
  - Zarządzanie klastrami kontenerów (Docker Swarm vs Kubernetes)
- **Sieć w Dockerze:**
  - Zarządzanie overlay networks, service discovery