ROZPROSZONE SYSTEMY OPERACYJNE

PROJEKT

FAZA I

Zespół Mechatroniczni Wojownicy:
Dżumaga Rafał,
Grudzień Ewelina,
Gwiazdowicz Artur,
Jóźwik Mateusz,
Misiowiec Piotr,
Poćwierz Maciej,
Stepek Katarzyna

Politechnika Warszawska

2016-04-28

Spis treści

FAZA I

W ramach fazy I należało wykonać następujące zadania:

- Opis rozwiązania Docker
- Uruchomienie dostępnych w sieci wybranych konfiguracji demonstrujących wykorzystanie Docker i opisanie uruchomień w raporcie
- Koncepcja i architektura rozwiązania własnego opisana m.in. przez zestawienie wymagań funkcjonalnych i niefunkcjonalnych
- Harmonogram realizacji projektu z pełnym zdefiniowaniem i przydziałem ról w projekcie osadzonych w czasie realizacji projektu
- Organizacja środowiska programistycznego projektu

OPIS ROZWIĄZANIA DOCKER

Czym jest kontener

Kontener jest to środowisko uruchomieniowe zawierające aplikację wraz z dependencjami wymaganymi do jej poprawnego uruchomienia. Kontener oprócz samej aplikacji zawiera wyizolowaną instancję systemu operacyjnego, w której działa aplikacja. Celem stworzenia kontenerów było wyizolowanie aplikacji od zasobów systemu operacyjnego tak, aby miała dostęp tylko do wydzielonej jego części. Kontenery mają wiele wspólnych cech z maszynami wirtualnymi, lecz głównym celem wirtualnych maszyn jest pełna symulacja środowiska uruchomieniowego, natomiast kontenery umożliwiają przenośność aplikacji. Ich głównymi cechami są:

1. Współdzielenie zasobów z systemem operacyjnym, w którym się znajdują (tzw. Host OS). Dzięki temu jest to wysoce wydajne rozwiązanie, pozwalające na uruchamianie aplikacji znajdujących się w kontenerze w kilka sekund.
2. Kontenery są odporne na zmiany konfiguracyjne środowiska, w którym się znajdują.
3. Mała ilość pamięci, jaką zajmuje kontener, pozwala na uruchomienie wielu instancji w tym samym czasie w celu np. zasymulowania środowiska produkcyjnego rozproszonego systemu.
4. Użytkownicy mogą w łatwy sposób uruchomić stworzone przez innych kontenery i aplikacje bez potrzeby czasochłonnego konfigurowania środowiska.

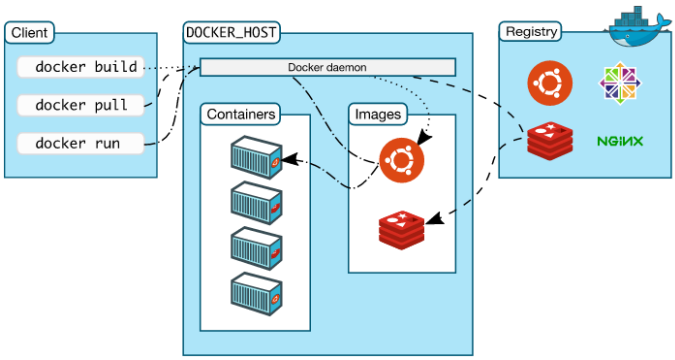
Czym jest Docker

Docker jest rozwiązaniem bazującym na technologii kontenerów, jednak w wielu aspektach poprawia i ułatwia pracę z nimi m.in. dzięki przenośnym obrazom kontenerów, czy też przyjaznym dla użytkownika interfejsie, umożliwiającym tworzenie i rozpowszechnianie kontenerów.

Dwa podstawowe elementy Dockera to silnik (Docker engine) pozwalający na szybkie i łatwe uruchamianie kontenerów oraz Docker Hub – serwis w chmurze pozwalający na dystrybucję kontenerów.

Architektura Dockera

Docker działa w oparciu o architekturę klient – serwer. Klient komunikuję się z Docker deamonem, odpowiedzialnym za tworzenie, uruchamianie oraz dystrybucję kontenerów. Obsługuje również polecenia użytkownika i przekazuje je do Docker deamona. Oba komponenty mogą być uruchomione w jednym systemie lub klient może komunikować się ze zdalnym deamonem.



Źródło: https://docs.docker.com/engine/article-img/architecture.svg

Podstawowe pojęcia

Aby w pełni zrozumieć jak działa Docker, należy zapoznać się z trzema pojęciami:

1. Obraz (Docker image)
2. Rejestr (Docker registry)
3. Kontener (Docker container)

Obraz

Jest to obraz stworzonego kontenera. Obraz może zawierać np. system Ubuntu oraz aplikację stworzoną przez użytkownika. Docker umożliwia w prosty sposób tworzenie obrazów lub aktualizację już istniejących, a także korzystanie z obrazów stworzonych przez innych użytkowników. Z każdego obrazu można uruchomić instancję kontenera a następnie po wprowadzeniu pożądanych zmian stworzenie nowej wersji obrazu.

Jak zbudowany jest obraz

Każdy obraz składa się kilku warstw (Ang. Layers), które przy wykorzystaniu UFS (Union File System) łączone są w pojedynczy obraz. UFS pozwala plikom i folderom z różnych systemów plików pokrywać się, tworząc jeden spójny system plików. Dzięki warstwom Docker jest rozwiązaniem zajmującym mało miejsca w pamięci. Przy modyfikacji obrazu Dockera, np. poprzez aktualizację aplikacji do nowszej wersji, tworzona jest nowa warstwa zawierająca aplikację a nie cały nowy obraz. Dzięki temu nie ma potrzeby dystrybucji całego obrazu, a jedynie aktualizacji.

Rejestr

Służy do przechowywania obrazów dockera. Istnieją zarówno prywatne, jak i publiczne rejestry. Oficjalny rejestr to Docker Hub, który przechowuje ogromną ilość obrazów zarówno oficjalnych, stworzonych przez np. Ubuntu czy Microsoft, jak również i takich stworzonych przez użytkowników.

Kontener

Kontener zawiera wszystkie elementy niezbędne do uruchomienia aplikacji. Każdy kontener jest stworzoną z obrazu wyizolowaną i bezpieczną platformą uruchomieniową dla aplikacji.

Jak działa kontener

Kontener składa się z systemu operacyjnego, plików dodanych przez użytkownika oraz meta danych. Każdy kontener zbudowany jest z obrazu który zawiera informacje o swojej zawartości, procesach które zostaną uruchomione przy starcie kontenera oraz pliki konfiguracyjne potrzebne do poprawnego wystartowania. Docker uruchamiając kontener z obrazu odczytuje te dane, a następnie dodaje do obrazu warstwę *read-write,* w której uruchamiana jest aplikacja.

Docker Toolbox

W celu zainstalowania Dockera na systemie operacyjnym Windows 10 należy skorzystać z jakiejś formy wirtualizacji. Możliwe jest zainstalowanie maszyny wirtualnej np. Linux Ubuntu 14.04 i uruchomienie w niej Dockera lub skorzystanie z rozwiązania zaoferowanego przez producenta czyli z tzw. Docker Toolboxa. Składa się on z następujących komponentów:

1. Maszyna (docker machine) – umożliwiająca korzystanie z komend *docker-machine*.
2. Silnik (docker enging) – umożliwiający korzystanie z komend *docker*.
3. Docker compose – umożliwiający korzystanie z komend *docker-compose*.
4. Kinematic – interfejs GUI
5. Powłoka skonfigurowana do korzystania z dockera
6. Oracle virtual-box

Maszyna Dockera

Jest to narzędzie pozwalające na instalację silnika dockera na wirtualnych hostach oraz zarządzanie nimi z poziomu konsoli użytkownika. Maszyna pozwala na tworzenie lokalnych oraz zdalnych hostów np. przy wykorzystaniu rozwiązań takich jak Amazon Web Services lub Digital Ocean. Maszyna dockera jest to jedyne rozwiązanie pozwalające na tę chwilę na korzystanie z Dockera na systemie Windows oraz najlepsze narzędzie dzięki któremu możemy zarządzać na raz wieloma instancjami hostów dockera (zarówno lokalnymi, jak i zdalnymi).

Źródło: https://docs.docker.com/machine/img/machine.png

Docker compose

Jest to narzędzie pozwalające w łatwy sposób tworzyć i uruchamiać tzw. *multi-container Docker applications*. W tym celu tworzony jest specjalny plik konfiguracyjny opisujący wszystkie wystawione serwisy, który następnie zostaje odczytany podczas uruchamiania środowiska za pomocą jednej komendy.

Docker compose pozwala w łatwy sposób na:

1. Budowanie obrazów dockera
2. Uruchomienie kontenerowych aplikacji jako serwis
3. Uruchomienie całego systemu jako serwis
4. Zarządzenie stanem konkretnych serwisów w systemie
5. Skalowanie serwisów

Docker compose nie skupia się na jednym kontenerze ale pozwala na opisanie całego środowiska oraz interakcji między serwisami (kontenerami).

Przykładowy plik konfiguracyjny (`docker-compose.yml`):

```
# Filename: docker-compose.yml
wordpress:                          (1)
 image: wordpress:4.2.2             (2)
 links:
  - db:mysql                        (3)
 ports:
  - 8080:80                         (4)

db:                                 (5)
 image: mariadb                     (6)
 environment:
  MYSQL_ROOT_PASSWORD: example      (7)
```

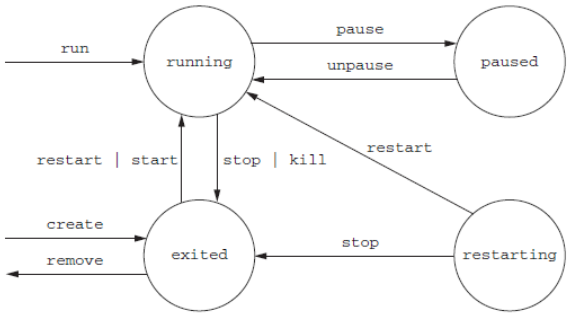
(1) Zdefiniowanie serwisu o nazwie: ‘wordpress’
(2) Skorzystanie z oficjalnego obrazu ‘wordpress:4.2.2’
(3) Linkowanie dependencji
(4) Mapowanie portu 80 kontenera do portu 8080 hosta
(5) Zdefiniowanie serwisu o nazwie: ‘db’
(6) Skorzystanie z oficjalnego obrazu mariadb:latest
(7) Zdefiniowanie zmiennej środowiskowej o nazwie MYSQL_ROOT_PASSWORD

Polecenia docker-compose:

a. `docker-compose ps` – wyświetla informacje o kontenerach uruchomionych plikiem *.yml
b. `docker-compose rm -v` – usuwa wszystkie kontenery uruchomione danym plikiem *.yml, flaga –v usuwa dodatkowo wolumeny połączone z kontenerami
c. `docker-compose up -d` – uruchamia środowisko zdefiniowane pliku *.yml w trybie detached
d. `docker-compose logs` – wyświetla logi wszystkich kontenerów
e. `docker-compose logs` name1 name2 – wyświetla logi danych kontenerów
f. `docker-compose pull` pobiera obrazy potrzebne do uruchomienia danego środowiska

Możliwe stany kontenera

Diagram przedstawiający wszystkie stany w który może znaleźć się kontener oraz komendy które zmieniają stan kontenera:



Źródło: Docker in Action. Jeff Nickoloff. Manning 2016

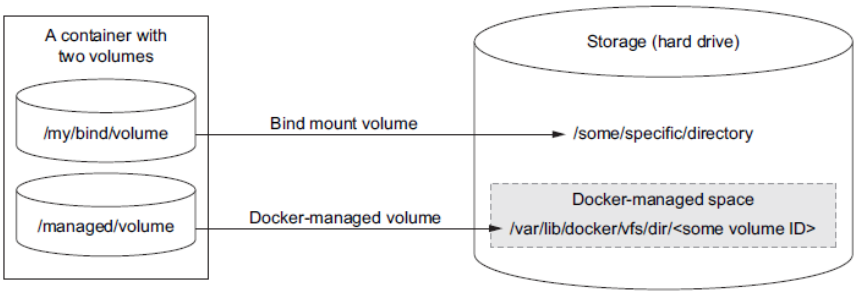
Docker PID Namespace

Każdy proces uruchomiony w systemie Linux posiada swój unikalny identyfikator, tzw. PID. Przestrzeń PID (PID namespace) jest zbiorem możliwych liczb identyfikujących proces. Podczas tworzenie kontenera Docker tworzy również dla niego przestrzeń PID. Bez osobnej przestrzeni PID dla każdego kontener procesy z kilku kontenerów posiadałyby te same identyfikatory lub identyfikatory procesów hosta. Ponadto procesy jednego kontenera byłyby w stanie ingerować w działanie procesów innego kontenera. Brak przestrzeni PID powodowałby również konflikty w dostępie do zasobów kilku kontenerów ponieważ starałby się korzystać z tego samego zasobu.

Docker Volumes

Wolumen (Volume) to element (tzw. mount point) drzewa katalogów kontenera, w którym część drzewa folderów hosta została zamontowana. Istnieją dwa typy wolumenów:

1. Bind Mount – korzystają ze ścieżki dostępu zdefiniowanej przez użytkownika. Przydatny, gdy host dostarcza plik lub folder, który musi być zamontowany w konkretnym miejscu kontenera. Umożliwia współdzielenie danych z innymi procesami uruchomionymi np. w systemie operacyjnym hosta. Możliwe jest również dzielenie wolumenu przez kilka różnych kontenerów (np. jeden zapisuje dane do wspólnego wolumenu drugi je odczytuje).
2. Korzystają ze ścieżki stworzonej przez daemona Dockera w kontrolowanej przez niego przestrzeni tzw. *Docker Managed Space*.



Źródło: Docker in Action. Jeff Nickoloff. Manning 2016

Ważną cechą wolumenów jest to, że nie są usuwane wraz z kontenerami, które z nich korzystają, jeśli tego nie zażądamy. Dlatego też, aby uniknąć problemów spowodowanych ograniczeniem pamięci przez tzw. „sieroce wolumeny", należy skorzystać z flagi –v przy usuwaniu ostatniego kontenera, który korzysta z danego wolumenu. Dzięki temu, usunięte zostaną również wszystkie powiązane wolumeny.

Logowanie

Docker automatycznie loguje wszystkie dane przesłane do STDOUT i STDERR. Logi następnie mogą zostać wyświetlone przy użyciu komendy: `docker logs`. Istnieje kilka metod logowania które mogą zostać włączone poprzez dodanie argumentu `--log-driver` do komendy startującej kontener. Są to między innymi: `json-file, syslog, journald, gelf, fluentd.`

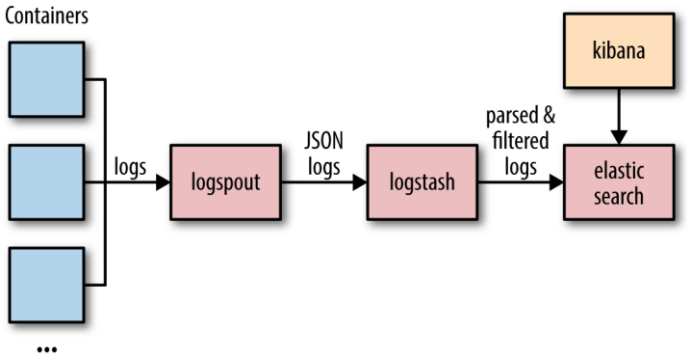
Żaden z tych driverów nie umożliwia jednak logowania w środowisku multihost. Możliwa jest jednak agregacja logów poprzez:

1. Uruchomienie w każdym kontenerze procesu, który działa jako agent zbierający logi i wysyłający je do serwisu agregacyjnego.
2. Zbieranie logów na hoście albo innym kontenerze specjalnie do tego przeznaczonym, a następnie wysyłanie ich do serwisu agregacyjnego.

Przykładowe rozwiązanie wykorzystuje drugi sposób. W tym celu mogą zostać użyte następujące narzędzia:

a. Elasticsearch – jest to wyszukiwarka działająca prawie w trybie rzeczywistym. Jest zaprojektowana tak, aby łatwo można było ją skalować na wielu węzłach więc jest to bardzo dobre rozwiązanie do przeszukiwania ogromnych ilości logów.
b. Logstash – narzędzie służące do odczytywania, parsowania oraz filtrowania logów przed wysłaniem do innego serwisu (np. Elasticsearcha)
c. Kibana – Graficzny interfejs do Elasticsearcha.
d. Logspout – narzędzie dzięki któremu w prostu sposób można przesłać logi z Dockera do Logstasha
e. Logspout-logstash adapter – mapuje logi z formatu dockera na format łatwo odczytywany przez logstasha

Schemat całego rozwiązania:



Źródło: Docker in Action. Jeff Nickoloff. Manning 2016

Docker Swarm

Docker Swarm (rój) to rozwiązanie klastrowe zaimplementowane przez Dockera. Pozwala ono traktować grupę hostów jako pojedynczego hosta.

W celu stworzenia klastra należy pobrać obraz 'Swarm', a następnie korzystając z Dockera skonfigurować managera oraz wszystkie hosty.

Konfiguracja obejmuje otworzenie portu TCP oraz zainstalowanie Dockera na każdym węźle oraz stworzenia i skonfigurowania certyfikatów TLS w celu zabezpieczenia klastra.

Obraz Swarm jest oficjalnym obrazem wspieranym i aktualizowanym przez Dockera. Istnieją również inne metody tworzenia klastra, lecz stworzenie go przy pomocy kontenera Swarm jest najprostszą z metod, wymagającą jedynie uruchomienia komendą `'docker run'`. Ponadto kontener izoluje klaster od środowiska hosta.

Bezpieczeństwo

Kontenery ustępują pod względem bezpieczeństwa wirtualnym maszyną, a ich niektórymi problemami są:

1. **Współdzielenie jądra systemu między wszystkimi kontenerami i hostem**
   Po wykryciu wewnętrznego błędu jądra systemu uszkodzone mogą zostać wszystkie kontenery oraz host. W przypadku maszyn wirtualnych atakujący aby dostać się do jądra systemu, musiałby przedostać się zarówno przez jądro maszyny jak i przez hypervisora.
2. **Ataki DoS**
   Kontenery współdzielą między sobą zasoby jądra, przez co po przejęciu dostępu do zasobów takich jak np. pamięć operacyjna czy UID, inne kontenery mogą zostać zagłodzone, co prowadzi do niedostępności seriwsu (Denial of Service).

W celu zabezpieczenia się przed atakami należy:

1. Korzystać z load balancerów lub kontenerów proxy w celu ograniczenia ilości kontenerów, które otwierają swoje porty na świat.
2. Należy maksymalnie ograniczać prawa użytkowników
3. Należy korzystać z kontenerów monitorujących oraz logujących
4. Aktualizować oprogramowanie kontenerów
5. Korzystać z zabezpieczeń AppArmor lub SELinux jeśli to możliwe
6. Uruchamiać pliki systemowe jako *read-only* jeśli to możliwe
7. Ograniczać pamięć dostępną dla kontenera korzystając z flagi `-manvolnum`
8. Szyfrować komunikację między kontenerami.
9. Korzystać z polityki restartowania `on-failure` zamiast `always` w celu ograniczenia wykorzystywania zasobów (możliwe przy ataku DoS)
10. Ograniczać dostęp kontenerów do zasobów CPU
11. Otwierać tylko wykorzystywane porty

RAPORT Z TESTÓW DOCKERA

W ramach testów zostały przeprowadzone trzy scenariusze

1. Aplikacja działająca na jednym kontenerze
2. Środowisko składające się z trzech kontenerów
3. Środowisko klastrowe składające się z trzech kontenerów na trzech różnych maszynach dockera

1. Test pojedynczego kontenera

W celu uruchomienia aplikacji znajdującej się w repozytorium Dockera korzystamy z komendy:

```
docker run dockerinaction/hello_world
```

Pierwsze dwa elementy komendy przekazują informację Deamonowi Dockera, potrzebną do uruchomienia kontenera. Ostatni człon to nazwa repozytorium, które zostanie pobrane. Po uruchomieniu komendy Docker wykonuje następujące kroki:

1. Szuka obrazu na naszym komputerze
2. Jeśli go nie znalazł, przeszukuje repozytorium (Docker Hub)
3. Jeśli obraz znajduje się w repozytorium Docker, pobiera go
4. Instaluje warstwy obrazu na hoście uruchomieniowym
5. Tworzy nowy kontener i startuje w nim aplikację

2. Test współpracujących kontenerów

Testowane środowisko składa się z trzech komunikujących się ze sobą kontenerów. Na pierwszym kontenerze uruchomiona zostanie instancja serwera *Nginx*, na drugim program wysyłający emaile, obydwa kontenery zostaną uruchomione w trybie *detach*. Ostatni kontener jest kontenerem interaktywnym, dzięki któremu użytkownik może komunikować się z pozostałymi. Pierwszym krokiem jest wykonanie komendy:

```
docker run --detach --name web nginx:latest
```

Pobiera ona i uruchamia najnowszą wersję serwera *Nginx* z repozytorium dockera. Outputem komendy jest unikalny identyfikator kontenera.

- Flaga `--detach` (opcjonalnie *–d*) uruchamia kontener w tle. Oznacza to, że program został uruchomiony ale nie został podpięty do konsoli użytkownika. Następnie uruchamiamy kolejny kontener w tle. Jest on odpowiedzialny za przyjmowanie zgłoszeń i wysłanie maili:

```
docker run -d --name mailer dockerinaction/ch2_mailer
```

Ostatni kontener potrzebny do zrealizowania zadania jest kontenerem interaktywnym.

```
docker run --interactive --tty --link web:web --name web_test
busybox:latest /bin/sh
```

- Flaga `--interactive` (lub *–i*) otwiera strumień wejściowy dla kontenera
- Flaga `--tty` alokuje wirtualny terminal dla dockera.
- Flaga `--link web:web` powiązała nowo utworzony kontener z kontenerem web

Po zakończeniu komendy Docker wystartował program powłowki: *sh*. Wszystkie komendy które zostaną od tej pory wpisane, są komendami działającymi tylko na danym kontenerze (czyli nie możemy już stosować komendy docker run…) Aby sprawdzić poprawne linkowanie kontenerów korzystamy z komendy `wget` wysyłające zapytanie http do serwera:

```
wget -O - http://web:80/
```

Komenda `exit` pozwala nam opuścić kontener. Ostatni element rozwiązania to agent monitorujący działanie serwera. Aby go uruchomić:

```
docker run -it --name agent --link web:insideweb --link mailer:insidemailer dockerinaction/ch2_agent
```

Kontener nie został uruchomiony w trybie *detach* dlatego, aby odłączyć konsolę od kontenera, należy przytrzymać ctrl, nacisnąć klawisz p, a potem q. W celu potwierdzenia uruchomienia wszystkich trzech kontenerów można skorzystać z komendy:

`docker ps` – wyświetla ona podstawowe informacje o wszystkich działających kontenerach.

W celu zrestartowania któregoś z kontenerów korzystamy z komendy:

```
docker restart name (np. docker restart mailer)
```

Aby wyświetlić logi kontenera możemy skorzystać z komendy:

```
docker logs name (np. docker logs agent).
```

Po przejrzeniu wszystkich logów można zatrzymać kontener web:

```
docker stop web
```

a następnie sprawdzić logi kontenera ‘mailer’:

```
docker logs mailer
```

Widać, że kontener agent wykrył zatrzymanie serwera web i powiadomił o tym kontener mailer.

3. Test tworzenia klastra

Na początku należy stworzyć token dla klastra korzystając z komendy `'swarm create':`

```
SWARM_TOKEN=$(docker run swarm create)
```

Posiadając token możliwe jest stworzenie hosta zarządzającego (mastera):

```
docker-machine create -d virtualbox \
--engine-label dc=a \
--swarm --swarm-master \
--swarm-discovery token://$SWARM_TOKEN \
swarm-master
```

Za pomocą tej komendy została stworzona nowa maszyna wirtualna o nazwie: swarm-master. Powiązana jest z klastrem za pomocą wcześniej utworzonego tokena. Korzystając z flagi `'--engine-label'`, oznaczyliśmy ją etykietą 'dc=a'. Następnie należy stworzyć dwie pozostałe maszyny wchodzące w skład klastra: swarm-1 i swarm-2.

```
docker-machine create -d virtualbox \
--engine-label dc=a \
--swarm \
--swarm-discovery token://$SWARM_TOKEN \
swarm-1

docker-machine create -d virtualbox \
--engine-label dc=b \
--swarm \
--swarm-discovery token://$SWARM_TOKEN \
swarm-2
```

Aby sprawdzić poprawność wykonanych komend możemy wylistować wszystkie maszyny przypisane do klastra o danym tokenie.

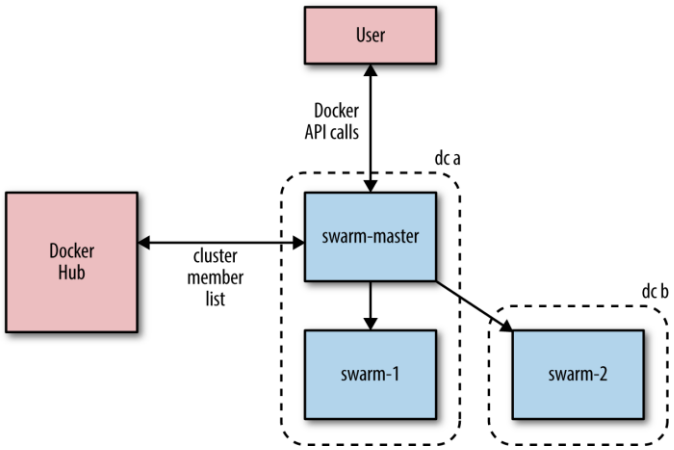
```
docker run swarm list token://$SWARM_TOKEN
```

W celu połączenia się z managerem klastra należy skorzystać z komendy:

```
eval $(docker-machine env --swarm swarm-master)
```

natomiast docker info wyświetla podstawowe informacje o klastrze

Architektura stworzonego rozwiązania:



Źródło: Docker in Action. Jeff Nickoloff. Manning 2016

KONCEPCJA I ARCHITEKTURA ROZWIĄZANIA

W ramach projektu realizowany będzie system do obsługi głosowania. Przyjmował on będzie dane z głosowania paczkami, zapisywał je do rozproszonej bazy danych, przetwarzał, a wyniki przetwarzania i analizy udostępniał użytkownikowi o odpowiednio wysokich uprawnieniach.

Role

Z systemu korzystałyby trzy rodzaje użytkowników:

- Użytkownik uprawniony do wprowadzania nowych danych
- Użytkownik uprawniony do oglądania wyników analizy danych
- Użytkownik zajmujący się utrzymaniem systemu i jego rekonfiguracją w razie potrzeby

Wymaganie funkcjonalne

FU.1. Odbieranie danych

FU 1.1: System zapewni możliwość importu danych w formacie JSON.
FU 1.2: Dane będą przesyłane do systemu asynchronicznie w postaci paczek.

FU.2 Przechowywanie danych

FU2.1: Składowanie odebranych danych oraz ich replikacja zapewniona przez warstwę bazy danych.

FU.3 Przetwarzanie danych

FU.3.1: Możliwość cyklicznego przetwarzania danych z bazy
FU 3.2: Możliwość podglądu otrzymanych wyników z poziomu aplikacji webowej.

FU.4 Administrowanie analizą danych

FU.4.1: Możliwość zarządzania procesem analizy danych.
FU.4.2: Możliwość określenia danych wyjściowych z poziomu aplikacji webowej.

FU.5 Kontrola dostępu

FU.5.1: Uzyskanie dostępu do systemu poprzedzane pomyślnym przejściem identyfikacji i uwierzytelniania. Uwierzytelnianie realizowane poprzez logowanie domenowe.
FU.5.2: Zapewnienie obsługi kilku rodzajów użytkowników, których uprawnienia zależne będą od ról przydzielonych w systemie.

Wymaganie niefunkcjonalne

NFU.1 Skalowalność

NFU.1.1 Baza danych przechowująca dane z głosowania oraz aplikacja 1 obsługująca ją uruchomione są na 3 węzłach. Liczba węzłów może być rozszerzona poprzez modyfikację konfiguracji systemu.

NFU.2 Niezawodność

NFU.2.1: Load Balancer znajduje się na 2 węzłach. W przypadku awarii głównego węzła pełniącego rolę load balancera, węzeł zapasowy (shadow) przejmuje jego rolę.
NFU.2.2: Aplikacja 1 (obsługująca rozproszoną zapis do rozproszonej bazy danych) uruchomiona jest na kilku węzłach na raz. W przypadku awarii jednego z nich, aplikacja będzie dostępna na pozostałych węzłach.
NFU.2.3: Baza danych przechowująca dane z głosowania jest bazą rozproszoną, znajdującą się na minimalnie 3 węzłach. Dodatkowo dane zapisywane są do niej w sposób redundantny. W ten sposób spójność danych jest zapewniona, dopóki choć jeden węzeł działa.
NFU.2.4: Aplikacja 2 oraz aplikacja 3 znajdują się na 2 węzłach. W przypadku awarii głównego węzła, na którym uruchomione są te aplikacje, węzeł zapasowy (shadow) staje się aktywny i przejmuje jego rolę.

NFU.3 Wydajność

NFU.3.1: Dużej wielkości dane wejściowe dzielone będą na mniejsze paczki i obługiwane przez różne węzły
NFU.3.2: Rozdzielenie sprzętowe aplikacji do zapisu danych wejściowych od aplikacji przetwarzającej te dane
NFU.3.3: Cachowanie danych aplikacji webowej w przeglądarce i oddelegowanie do niej części obliczeń.

NFU.4 Bezpieczeństwo

NFU.4.1: Wszystkie przechowywane dane chronione poprzez mechanizmy uwierzytelniania użytkowników.

NFU.5 Przenośność

NFU.5.1: Zapewnienie działania systemu niezależnie od środowiska uruchomieniowego poprzez użycie kontenerów umożliwiających enkapsulacje aplikacji wraz z ich dependencjami.

NFU.6 Dostępność

NFU.6.1: System powinien działać w trybie ciągłym
NFU.6.2: Awarie systemu powinny zostać usunięte przed upływem 4h od momentu zgłoszenia.
NFU.6.3: Czas trwania okien serwisowych nie powinien przekraczać 1h.

Architektura systemu

System będzie się składać z trzech aplikacji:

Aplikacja 1 – Odbiera od użytkownika nowe dane z głosowania i zapisuje do lokalnej bazy danych oraz do baz na innych węzłach tak, by zapewnić redundancję danych.

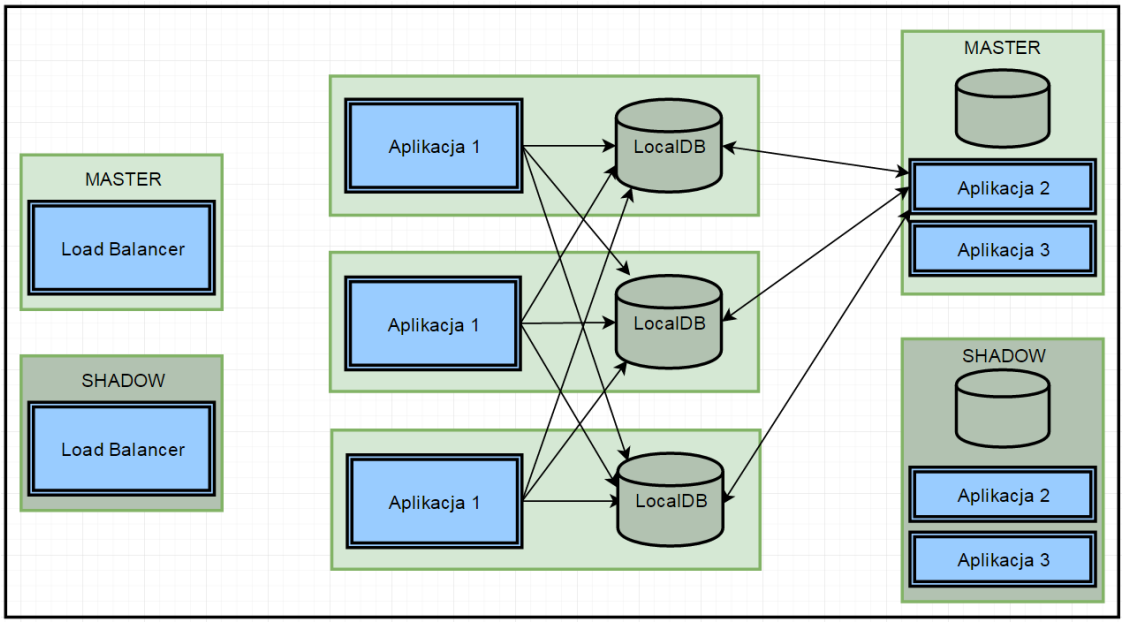
Aplikacja 2 – Cyklicznie przetwarza i analizuje dane zapisane przez aplikację 1, a wyniki analizy zapisuje w lokalnej bazie danych

Aplikacja 3 – Webowa aplikacja, która umożliwia użytkownikowi przeglądanie wyników analizy danych

Dostępność systemu będzie gwarantowana na dwa sposoby:

Aplikacja 1 uruchomiona będzie równolegle na kilku węzłach. Analogicznie, dane wejściowe będą zapisywane w kilku miejscach, żeby nie stracić do nich dostępu w momencie awarii jednego z węzłów

Aplikacja 2 i 3 uruchomione będą na jednym węźle, ale w chwili awarii inny przygotowany zawczasu węzeł go zastąpi. Podobnie będzie w przypadku Load Balancera.



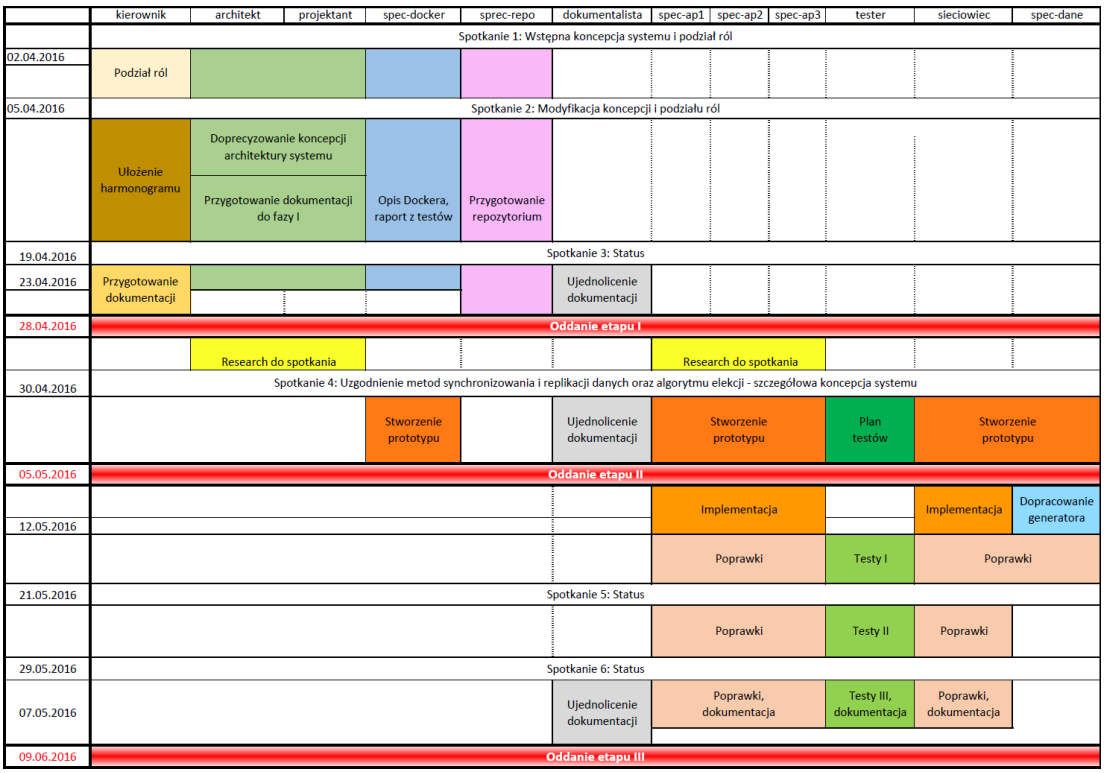
Wstępna koncepcja architektury systemu

HARMONOGRAM

Podział ról w zespole:

1. Kierownik – Katarzyna Stepek
2. Architekt – Artur Gwiazdowski, Maciej Poćwierz
3. Projektant – Ewelina Grudzień, Katarzyna Stepek
4. Spec od repozytorium – Artur Gwiazdowski
5. Dokumentalista – Katarzyna Stepek
6. Tester – Maciej Poćwierz
7. Handlowiec – Piotr Misiowiec
8. Spec od dockera – Mateusz Jóźwik
9. Programista aplikacji 1 - Artur Gwiazdowicz
10. Programista aplikacji 2 - Piotr Misiowiec
11. Programista aplikacji 3 – Rafał Dżumaga, Ewelina Grudzień
12. Spec od Load balancera i https – Mateusz Jóźwik
13. Generowanie danych wejściowych – Ewelina Grudzień

Z powodu dużego rozmiaru, harmonogram został umieszczony w lepszej wersji w dodatkowym załączniku (plik harmonogram.pdf)



	kierownik	architekt	projektant	spec-docker	sprec-repo	dokumentalista	spec-ap1	spec-ap2	spec-ap3	tester	sieciowiec	spec-dane
	Spotkanie 1: Wstępna koncepcja systemu i podział ról											
02.04.2016	Podział ról											
05.04.2016	Spotkanie 2: Modyfikacja koncepcji i podziału ról											
	Ułożenie harmonogramu	Doprecyzowanie koncepcji architektury systemu		Opis Dockera, raport z testów	Przygotowanie repozytorium							
		Przygotowanie dokumentacji do fazy I										
19.04.2016	Spotkanie 3: Status											
23.04.2016	Przygotowanie dokumentacji					Ujednolicenie dokumentacji						
28.04.2016	Oddanie etapu I											
		Research do spotkania					Research do spotkania					
30.04.2016	Spotkanie 4: Uzgodnienie metod synchronizowania i replikacji danych oraz algorytmu elekcji - szczegółowa koncepcja systemu											
				Stworzenie prototypu		Ujednolicenie dokumentacji	Stworzenie prototypu			Plan testów	Stworzenie prototypu	
05.05.2016	Oddanie etapu II											
12.05.2016							Implementacja				Implementacja	Dopracowanie generatora
							Poprawki			Testy I	Poprawki	
21.05.2016	Spotkanie 5: Status											
							Poprawki			Testy II	Poprawki	
29.05.2016	Spotkanie 6: Status											
07.05.2016						Ujednolicenie dokumentacji	Poprawki, dokumentacja			Testy III, dokumentacja	Poprawki, dokumentacja	
09.06.2016	Oddanie etapu III											

Organizacja środowiska programistycznego projektu

Na potrzeby projektu stworzono repozytorium i umieszczone je na portalu github.com pod adresem:

https://github.com/veiar/rso_voting

Dla bezpieczeństwa, specjalista od repozytorium będzie przechowywał aktualną kopię zapasową projektu.

Bibliografia:

[1] Using Docker. Adrian Mouat. O'Reilly 2016.

[2] Docker in Action. Jeff Nickoloff. Manning 2016.

[3] https://docs.docker.com/