Wyższa
Szkoła
Informatyki
i Umiejętności

WYŻSZA SZKOŁA INFORMATYKI I UMIEJĘTNOŚCI
Wydział Informatyki i Zarządzania
Kierunek: Informatyka

Sebastian Florek
nr albumu: 29571

Praca Magisterska
Monitorowanie otoczenia za pomocą mikrokontrolerów Raspberry Pi w oparciu o system zarządzania klastrem Kubernetes

Praca napisana pod kierunkiem
dr inż. Grzegorz Zwoliński

Rok akademicki 2016/2017

Spis treści

Rozdział 1

Wstęp

1.1 Problematyka i zakres pracy

Wraz z rozwojem Internetu Rzeczy na świecie powstają coraz to nowe urządzenia mające na celu automatyzację działań i ułatwienie życia człowieka. Dzięki ich zdolności do wzajemnej komunikacji, wymiany informacji oraz zdalnego zarządzania zasobami stają się one coraz bardziej popularne, a wręcz wymagane w życiu codziennym coraz większej grupy osób. Sterowanie oświetleniem, radiem czy innymi sprzętami elektronicznymi za pomocą naszego smartfona już nikogo nie dziwi. W wielu przypadkach urządzenia te muszą zbierać bardzo duże ilości danych oraz przesyłać je do centralnego punktu. Powoduje to ogromny wzrost ilości danych, które nie są w stanie zostać obsłużone przez jeden serwer. Powstaje więc potrzeba stworzenia niezawodnych, wydajnych i bezpiecznych systemów o wysokiej dostępności.

Technologie wirtualizacji [1] powstały w celu realizacji tych wymagań. Wirtualizacja serwerów dawno już wyparła tradycyjne serwery, które zostały zastąpione przez rozwiązania chmurowe. Niezależność sprzętowa, lepsza utylizacja zasobów, większe bezpieczeństwo, łatwa migracja danych i redukcja kosztów to tylko niektóre z wielu zalet wirtualizacji. Właśnie ta niezależność sprzętowa pozwala na coraz lepsze wykorzystanie mikrokontrolerów, których głównymi zaletami są mały koszt i niewielki pobór mocy, a dzięki coraz lepszej optymalizacji systemów i postępującej miniaturyzacji, również rosnąca wydajność.

Obecnie nie trzeba już wydawać ogromnych kwot w celu uruchomienia własnego klastra do zarządzania danymi, czy wyposażenia naszego mieszkania, a nawet biura w inteligentne czujniki i kontrolery. Wystarczą nam w tym celu mikrokontro-

[1] [?]

lery Raspberry Pi, które dzięki systemom typu open source takim jak Kubernetes pozwolą nam na stworzenie w pełni funkcjonalnego klastra.

Proponowanym rozwiązaniem powyższych problemów będzie projekt o nazwie KubePi. Projekt ten skupia się na wirtualizacji opartej o kontenery Dockera [2] zarządzane przez system zarządzania klastrem Kubernetes i ma na celu stworzenie rozproszonego systemu służącego do monitorowania otoczenia. Przykładowy klaster opierać się będzie na dwóch mikrokontrolerach Raspberry Pi. Do pierwszego urządzenia będącego zarazem głównym węzłem klastra podłączone zostaną trzy czujniki: temperatury, wilgotności oraz alkoholu. Jego zadaniem będzie udostępnianie zbieranych informacji. Drugie urządzenie zostanie natomiast wyposażone w wyświetlacz LED [3] , co pozwoli na odczyt i wyświetlanie temperatury raportowanej przez pierwsze urządzenie. Dodatkowo w klastrze zostanie zainstalowana aplikacja webowa [4] pozwalająca na zdalny monitoring. W celu lepszego zobrazowania komunikacji między urządzeniami zostanie ona uruchomiona na drugim urządzeniu.

1.2 Uzasadnienie wyboru tematu

W tej części opisane zostaną powody wyboru tematu związanego z systemem monitorowania otoczenia przy użyciu nowoczesnych technologii w oparciu o mikrokontrolery. Głównym powodem jest wykazanie użyteczności systemów chmurowych wysokiej dostępności w oparciu o mikrokontrolery. Wymaga to gwarancji pracy systemu nawet w razie uszkodzenia jednego z węzłów klastra. Drugim powodem jest zbadanie możliwości mikrokontrolerów jako alternatywy dla standardowych systemów chmurowych oraz porównanie wydajności względem ceny przy budowaniu klastra. Ze względu na małe rozmiary i koszt mikrokontrolery takie jak Raspberry Pi mogą służyć jako tania alternatywa do budowy inteligentnego systemu dla naszego domu i nie tylko.

[2] [**?**]
[3] [**?**]
[4] [**?**]

1.3 Metoda badawcza

1.4 Przegląd literatury w dziedzinie

1.5 Układ pracy

Celem pracy jest zaproponowanie architektury i sprawdzenie w działaniu rozproszonego systemu wysokiej dostępności służącego do monitorowania otoczenia.

Rozdział pierwszy zawiera szczegółowy opis problemu. Zostają w nim przedstawione różne problemy związane z wydajnością oraz bezpieczeństwem tradycyjnych rozwiązań, wraz z opisem metod badawczych użytych do analizy tematu. Podsumowane zostają również główne założenia i cele pracy. Na koniec przeprowadzony zostaje przegląd literatury związanej z tematem, z naciskiem na kluczowe zagadnienia dotyczące wirtualizacji, rozwiązań chmurowych oraz mikrokontrolerów opartych na architekturze ARM, wraz z krótkim opisem użytych źródeł.

W rozdziale drugim przybliżona zostaje tematyka systemów zarządzania klastrami pod kątem ich wymagań, bezpieczeństwa oraz komunikacji sieciowej. Kolejnym krokiem jest dokładniejsze zapoznanie się z wirtualizacją, a konkretniej wirtualizacją opartą o kontenery Dockera, co pozwoli lepiej zrozumieć ideę pracy. Następnie po krótce przedstawione zostają tematyki związane z mikrokontrolerami oraz Internetem Rzeczy.

Kolejny rozdział opisuję fazę projektowania i implementacji projektu KubePi. Spisane zostają wymagania funkcjonalne aplikacji, a także ograniczenia projektowe. Wymienione i opisane zostają użyte technologie. Opisany zostaje proces konfiguracji urządzeń, sieci oraz systemu. Następnie wskazane zostają kluczowe punkty aplikacji wraz z kodem źródłowym i opisem. W kolejny kroku przechodzimy do fazy testów stworzonych aplikacji jak i całego systemu.

W podsumowaniu pracy opisane zostają słabe i mocne strony przedstawionego rozwiązania. Na podstawie uzyskanych wyników następuje ocena możliwości i przydatności zaproponowanego rozwiązania. Na końcu omówione zostają możliwe perspektywy rozwoju projektu.

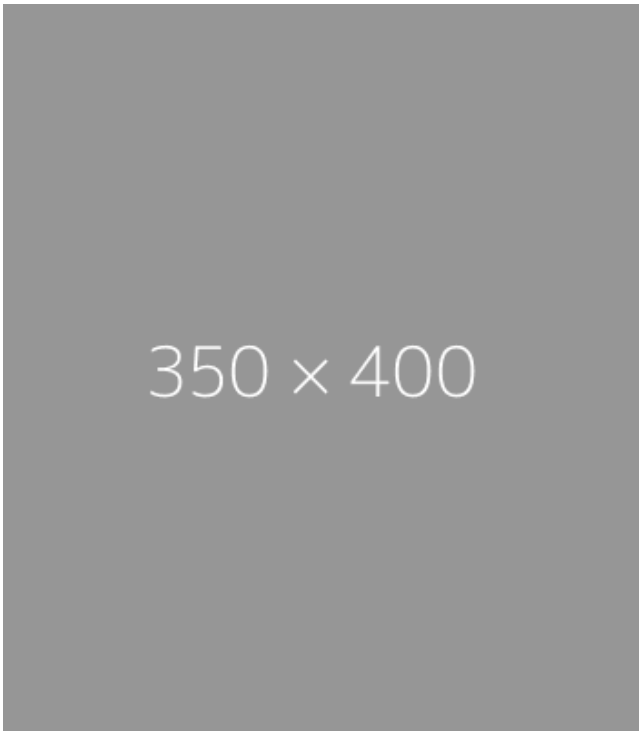
Rozdział 2

Podstawy teoretyczne

Użyte koncepcje i terminy używane w dalszej części pracy muszą zostać wyjaśnione w celu lepszego zrozumienia opisywanej problematyki. W kolejnych sekcjach zostają objaśnione podstawowe pojęcia związane z Internetem Rzeczy, przetwarzaniem danych w klastrze, tematyką wirtualizacji, rozproszonych systemów chmurowych oraz mikrokontrolerów. Następuje przedstawienie narzędzi do tworzenia, wdrażania i uruchamiania aplikacji rozproszonych w oparciu o kontenery aplikacji Docker. Następnie opisane zostaną systemy zarządzania kontenerami w klastrze takie jak Kubernetes czy trochę mniej zaawansowany Docker Swarm. Na koniec opisany zostanie system oparty o kontenery Dockera, który przenosi zalety kontenerów do Internetu Rzeczy, czyli Resin.io oraz kontroler Raspberry Pi wraz z interfejsami komunikacyjnymi użytymi przy tworzeniu tej pracy.

2.1 Internet Rzeczy

Internet Rzeczy zyskuje na popularności, głównie za sprawą swojego podejścia do komunikacji. Głównym założeniem jest umożliwienie urządzeniom oraz zwykłym przedmiotom codziennego użytku wzajemnej interakcji ze sobą jak i z użytkownikiem. Celem natomiast jest stworzenie inteligentnych urządzeń w celu zminimalizowania i jak największego uproszczenia interakcji użytkownika z takimi urządzeniami. Zaczynając od prostych czujników i wyświetlaczy a kończąc na autonomicznych pojazdach, budynkach czy całych miastach zarządzanych przez mikrokontrolery.



Rysunek 2.1: Przedstawienie Internetu Rzeczy w ujęciu graficznym

2.1.1 Obszary zastosowań

Należy zaznaczyć, że systemy realizacji Internetu Rzeczy są w zasadzie ograniczone tylko przez naszą wyobraźnię, a obejmują one między innymi:

- Inteligentne domy i budynki

  Interfejsy nowej generacji pozwolą nam nie martwić się o to czy zgasiliśmy światło, albo czy zamknęliśmy drzwi. System sam wykryje czy jesteśmy w domu lub pokoju i zrobi to za nas. Możliwość zarządzania urządzeniami codziennego użytku znacznie podwyższy nasz komfort i bezpieczeństwo.

- Inteligentne miasta

  Zaawansowane systemy pozwolą zoptymalizować infrastrukturę miejską. System sterowania ruchem dzięki zebranym danym sam dostosuje się do panujących warunków i pozwoli na rozładowanie ruchu w mieście. Sieć czujników może zostać użyta do wykrywania przestępstw czy sterowania oświetleniem w mieście.

- Inteligentne przedsiębiorstwa i przemysł

  Obecnie możemy doświadczyć dużych zmian w przedsiębiorstwach, które wykorzystują Internet Rzeczy w celu zarządzania całym łańcuchem produkcji i dostaw. Inteligentne maszyny same wiedzą w jakim są stanie i mogą reagować odpowiednio wcześnie w celu wymiany podzespołów czy przeprowadzenia konserwacji. Dla klienta natomiast dużym udogodnieniem mogą być zamówienia dostarczane przez drony, które są już testowane przez niektóre duże firmy takie jak Amazon czy UPS.

- Monitorowanie otoczenia i zagrożeń

  Rozległe sieci czujników już teraz pozwalają na całodobowe monitorowanie temperatury, opadów, wiatru, poziomu rzek, itp. Zebrane dane są przetwarzane i służą do wykrywania anomalii i przewidywania zdarzeń, które mogą zagrażać ludziom. W efekcie zwiększają one nasze ogólne bezpieczeństwo.

2.2 Chmury Obliczeniowe

Chmury obliczeniowe stają się coraz ważniejszym elementem Internetu Rzeczy. Dostarczają one swoje zasoby i usługi przy użyciu internetu. Słowo chmura opisuje sposób przetwarzania danych, który oderwany jest od naszego systemu, a przeprowadzany jest na zdalnych serwerach.

W szerszym ujęciu natomiast chmura jest centrum danych, w którym poszczególne węzły są wirtualizowane przy użyciu wirtualnych maszyn. Głównym powodem używania chmur obliczeniowych jest rozwiązywanie złożonych problemów oraz analiza dużych ilości danych. Wszystkie dane mogą być łatwo udostępniane innym osobom, a dostęp gwarantowany z każdego zakątka świata.

Bardziej formalna definicja chmur obliczeniowych przedstawia się następująco.

Chmura obliczeniowa 1 *Dostarczanie usług obliczeniowych, serwerów, magazynu, baz danych, sieci, oprogramowania analiz, itd. za pośrednictwem internetu. Firmy oferujące te usługi obliczeniowe są nazywane dostawcami chmury i zazwyczaj pobierają opłaty za usługi chmury obliczeniowej w zależności od użycia, podobnie jak dostawcy energii elektrycznej lub wody.*

2.2.1 Architektura

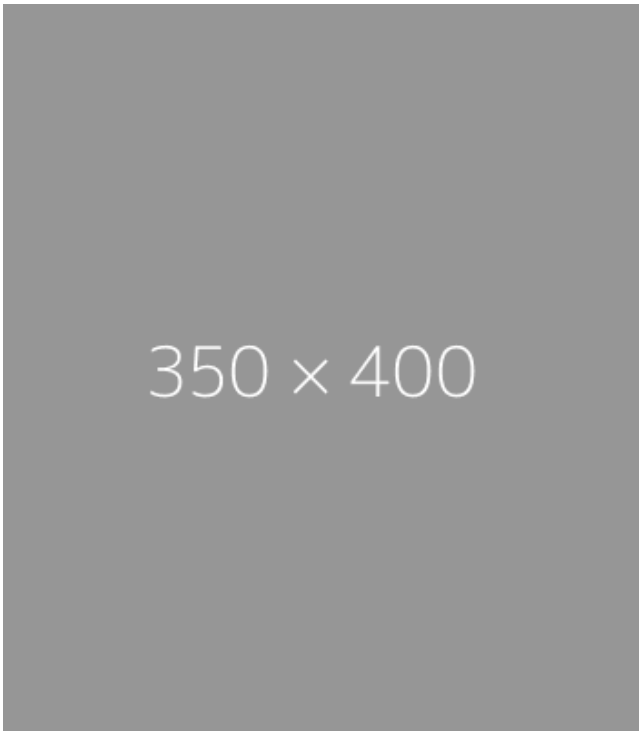
W związku z szybkim rozwojem chmur obliczeniowych, zostały one podzielone na trzy główne modele usług w celu utylizacji mocy obliczeniowej dla konkretnych potrzeb użytkowników. Modele te zostały zilustrowane na poniższym obrazku obrazującym architekturę chmur obliczeniowych. Architektura ta uwzględnia następujące podziały:

- Infrastruktura jako serwis (z ang. Infrastracture-as-a-Service) — znana również pod nazwą hardware-as-a-service[1]. Jest najprostszą odmianą usługi chmury obliczeniowej i pozwala na zdalny dostęp do zasobów sprzętowych. Dostarcza moc obliczeniową, przestrzeń dyskową i zasoby sieciowe. Jednym z najbardziej znanych dostawców takich usług jest Amazon Elastic Compute Cloud.

- Platforma jako serwis (z ang. Platform-as-a-Service) — usługa ta oferuje przede wszystkim środowisko deweloperskie oraz zbiór aplikacji, które mogą być używane do tworzenia i uruchamiania własnych aplikacji. Dostarcza bazy danych, serwery webowe, system operacyjny, środowisko uruchomieniowe, przestrzeń dyskową i wiele innych zasobów. Głównymi dostawcami tych usług są obecnie Microsoft Azure[2] oraz Google App Engine[3]

- Oprogramowanie jako serwis (z ang. Software-as-a-Service) — ostatni z modeli usług chmury obliczeniowej, który oferuje przechowywanie i uruchomienie aplikacji klienta na własnych serwerach oraz udostępnienie jej użytkownikom przez internet. Pozwala to na wyeliminowanie potrzeby instalacji oraz uruchamiania aplikacji na komputerze klienta. W efekcie to dostawca usług ma obowiązek zapewnienia ciągłości działania naszej aplikacji. Najbardziej znane usługi SaaS to np. Google Docs czy Google Apps.

[1] [?]
[2] [?]
[3] [?]



Rysunek 2.2: Architektura chmur obliczeniowych

2.3 Wirtualizacja

Ze względu na gwałtowny rozwój internetu, zapotrzebowanie na zasoby sprzętowe również wzrasta. Utylizacja i optymalne wykorzystanie zasobów zaczęły odgrywać kluczową rolę. Uruchamianie i zapewnienie ciągłości działania dużym aplikacjom sieciowym stworzyło potrzebę jak najlepszego zarządzania zasobami w celu maksymalnego wykorzystania dostępnych zasobów. Jednym z rozwiązań tych problemów jest właśnie wirtualizacja, czyli stworzenie wirtualnego środowiska zbudowanego w oparciu o infrastrukturę sieciową i dostarczenie go użytkownikom końcowym.

Wirtualizacja to tak na prawdę oddzielenie dostępnych zasobów od fizycznego sprzętu. Pozwala to uruchomić wiele systemów na tej samej platformie sprzętowej i systemowej w celu uzyskania jak najlepszej wydajności oraz utylizacji zasobów. Przy użyciu jednej platformy sprzętowej można dostarczyć usługi wielu użytkownikom końcowym. Dwoma najbardziej znanymi podejściami do wirtualizacji są: wirtualizacja oparta o nadzorcę oraz o kontenery. Zostaną one opisane w następnych sekcjach.

2.3.1 Wirtualizacja oparta o nadzorcę

Podejście to w ciągu ostatniej dekady było szeroko używane do tworzenia środowisk wirtualnych dla dużych systemów obliczeniowych. Nadzorca znany również jako monitor wirtualnej maszyny jest oprogramowaniem, które ma za zadanie tworzenie, monitorowanie wirtualnych środowisk oraz przydzielanie im zasobów. Przykładem może być uruchomienie systemu operacyjnego Windows jako maszyny wirtualnej na systemie operacyjnym Linux.

Dodatkowo wirtualizacja dostarcza nam niezależne środowisko, które może służyć do uruchamiania aplikacji w izolowanym środowisku bez ingerencji w inne aplikacje. Nadzorców możemy podzielić na dwa typy, które zostały zilustrowane na poniższym rysunku:

- Native/Bare metal hypervisor — ten nadzorca działa bezpośrednio na poziomie sprzętu. Dzięki pełnej kontroli nad sprzętem, może kontrolować i monitorować uruchomione systemy operacyjne, które działają poziom wyżej niż nadzorca. Popularnymi przykładami takich nadzorców są: Oracle VM, Microsoft Hyper-V czy VMWare ESX.
- Host based hypervisor — ten nadzorca działa jako program uruchomiony na danym systemie operacyjnym (hoście). Pełni on rolę emulatora i izoluje wirtualne systemy operacyjne od sprzętu jak i od hosta, czyli działa dwa poziomy ponad sprzętem. Popularnymi przykłądami takich nadzorców są: VirtualBox, VMWare Workstation, KVM.

Rysunek 2.3: Architektura chmur obliczeniowych

Właściwości nadzorcy

Według książki [odnosnik] nadzorcy wyróżniają się następującymi właściwościami:

1. Transparentność — pozwala uruchamiać oprogramowanie na maszynie wirtualnej bez żadnych modyfikacji i niezależnie od sprzętu oraz umożliwia dzielenie zasobów sprzętowych przez wiele maszyn wirtualnych.

2. Izolacja — umożliwia nadzorcy tworzenie i uruchamianie niezależnych, odizolowanych od siebie środowisk wirtualnych w obrębie jednego fizycznego systemu. Główną zaletą jest zapobieganie oddziaływania błędów aplikacji na inne aplikacje uruchomione w innych środowiskach wirtualnych.

3. Enkapsulacja — zapewnia elastyczność i bezpieczeństwo aplikacji uruchomionych w środowisku wirtualnym poprzez zamknięcie systemu w obrębie wirtualnego dysku twardego. Dzięki temu instalacja i tworzenie kopii zapasowych maszyn wirtualnych sprowadza się do kopiowania plików.

4. Łatwość zarządzania — wbudowane opcje nadzorcy do zamykania, restartowania, uruchamiania, usypiania, dodawania oraz usuwania wirtualnych maszyn umożliwiają łatwe zarządzanie wieloma maszynami wirtualnymi.

2.3.2 Wirtualizacja oparta o kontenery

Wirtualizacja oparta o kontenery jest mniej wymagającym w kontekście zasobów podejściem do wirtualizacji. Operuje ona na poziomie systemu operacyjnego i tworzy środowiska wirtualne jako procesy systemowe, co pozwala na dzielenie zasobów sprzętowych z systemem operacyjnym. Wprowadza ona pewien poziom abstrakcji, w którym jądro systemu jest dzielone pomiędzy kontenery i umożliwia uruchamianie więcej niż jednego procesu w obrębie każdego kontenera. Dzięki takiemu rozwiązaniu nie ma potrzeby uruchamiani pełnego systemu operacyjnego odizolowanego od systemu operacyjnego hosta. Przekłada się to bezpośrednio na oszczędność zużycia pamięci, czasu procesora i przestrzeni dyskowej.

Na poniższym rysunku możemy zobaczyć cztery kontenery uruchomione w systemie operacyjnym. Zasoby sprzętowe dzielone są pomiędzy nadrzędny system operacyjny oraz systemy operacyjne gości (kontenery). Zaletą tego typu wirtualizacji jest możliwość uruchomienia dużej ilości kontenerów w obrębie jednego systemu operacyjnego. Ma to również swoje wady, a mianowicie nie możemy przykładowo uruchomić systemu operacyjnego Windows jako kontener aplikacji na hoście z systemem operacyjnym Linux. Dodatkowo w porównaniu ze standardowymi nadzorcami kontenery nie gwarantują odpowiedniej izolacji zasobów, co

może mieć wpływ na bezpieczeństwo. Funkcje wirtualizacji przy użyciu kontenerów wykorzystują dwie podstawowe właściwości jądra systemu: grupy kontrolne(cgroups) oraz przestrzenie nazw(namespaces).

Rysunek 2.4: Architektura wirtualizacji opartej o kontenery

Grupy kontrolne

Grupy kontrolne to jedna z głównych funkcji jądra systemu, która pozwala użytkownikom alokować oraz ograniczać zasoby pomiędzy grupami procesów. Zasoby te to między innymi czas procesora, pamięć systemowa, przestrzeń dyskowa czy wykorzystanie sieci. Grupy kontrolne umożliwiają również sprawne zarządzanie zasobami przeznaczonymi dla kontenerów i działających w nich aplikacji. Przykładowo jeśli aplikacja tworzy dwa procesy z odrębnymi zasobami, może być rozdzielona na dwie grupy kontrolne z różnymi limitami użycia zasobów. Dodatkowo dzięki użyciu grup kontrolnych można w łatwy sposób monitorować i odmawiać lub przyznawać zasoby odpowiednim procesom. Możliwa jest również dynamiczna konfiguracja w trakcie działania systemu. Dzięki temu administrator może w łatwy sposób kontrolować zarządzanie zasobami w systemie.

Przestrzenie nazw

Przestrzenie nazw to kolejna ważna funkcjonalność dostępna w jądrze systemu, która pozwala na izolacje procesów wewnątrz kontenerów. Gwarantuje ona każdemu procesowi dedykowaną przestrzeń, w której może on działać, bez ingerencji w procesy uruchomione w innych przestrzeniach nazw. Dodatkowo zapewnia

izolację środowiska uruchomieniowego procesów. Obecnie w systemach Linux możemy znaleźć sześć domyślnych przestrzeni nazw:

- zamontowanych systemów plików (mnt) — zawiera drzewo katalogów z zamontowanymi systemami plików. Podstawowy system plików jest montowany w korzeniu drzewa w czasie uruchamiania systemu. Utworzenie osobnej przestrzeni nazw systemu plików dla grupy procesów umożliwia zamontowania sytemu plików, który będzie widoczny tylko dla tej grupy procesów.
- stacji roboczej i domeny (uts) — przechowuje aktualną nazwę stacji roboczej oraz nazwę domeny, do której należy stacja robocza. Utworzenie osobnej przestrzeni nazw UTS umożliwia zmianę tych parametrów dla poszczególnej grupy procesów.
- identyfikatorów procesów (pid) — umożliwia istnienie wielu procesów w różnych przestrzeniach nazw z tym samym identyfikatorem procesu. Użyteczna przy przenoszeniu grupy procesów między maszynami bez zmiany ich identyfikatorów.
- obiektów IPC (ipc) — pozwala na tworzenie systemowych obiektów do komunikacji między procesami, widocznych tylko dla grupy procesów, które współdzielą dana przestrzeń nazw IPC.
- użytkowników (user) — pozwala na odizolowanie procesów bazując na identyfikatorze użytkownika i grupy, która uruchomiła dany proces. Dzięki temu dany proces może być uruchomiony z przywilejami administratora w konkretnej przestrzeni nazw, natomiast w innych działać jako proces z prawami zwykłego użytkownika.
- zasobów sieciowych (net) — pozwala na odizolowanie podsystemu sieciowego dla grupy procesów. Dzięki temu grupa procesów może konfigurować urządzenia sieciowe oraz nawiązywać połączenia niezależnie od reszty systemu. Taka grupa posiada własne urządzenia sieciowe, adresy IP, trasy, itp.

Właściwości kontenerów

Według [dodac ksiazke], możemy wyróżnić cztery właściwości kontenerów Linuxa, które sprawiają, że są one atrakcyjne dla użytkowników:

1. Przenośność — kontenery mogą działać na wielu różnych środowiskach i nie wymagają dodatkowych kroków w celu dostosowania systemu operacyjnego. Dodatkowo można uruchamiać wiele aplikacji w obrębie jednego kontenera, a następnie uruchamiać je na różnych środowiskach.

2. Szybkość tworzenia — kontenery są łatwe i szybkie w budowie. W porównaniu z tworzeniem standardowej maszyny wirtualnej ich budowa i start zajmują sekundy, a nie minuty. Dzięki temu administratorzy mogą w łatwy sposób uruchamiać kontenery w środowisku produkcyjnym. Dodatkowo skracają czas potrzebny na stworzenie aplikacji uruchamianej w kontenerze. W łatwy sposób pozwalają na dzielenie się aplikacjami z zespołem i testowanie w różnych środowiskach.

3. Skalowalność — tworzenie i instalacja kontenerów na dowolnym systemie czy w chmurze obliczeniowej są bardzo proste. Dużym plusem jest również skalowalność kontenerów. W prosty sposób możemy zautomatyzować tworzenie czy usuwanie kontenerów bazując na aktualnym zużyciu zasobów sprzętowych. Dzięki temu idealnie nadają się one do instalacji w chmurze.

4. Elastyczność — przy użyciu kontenerów możemy w prosty sposób uruchomić dużą liczbę aplikacji na pojedynczym systemie operacyjnym. Biorąc pod uwagę, że nie uruchamiają one pełnego systemu operacyjnego, możemy w łatwy i wydajny sposób zarządzać wieloma aplikacjami.

Po zapoznaniu się z technikami wirtualizacji opartymi o nadzorcę i kontenery możemy w łatwy i czytelny sposób przedstawić ich porównanie w formie poniższej tabeli.

Rysunek 2.5: Architektura wirtualizacji opartej o kontenery

2.3.3 Docker

Docker jest jednym z najpopularniejszych narzędzi bazujących na wirtualizacji opartej o kontenery. Został on po raz pierwszy przedstawiony przez Solomona Hykesa, 15 marca 2013 roku. W tamtym czasie niewiele, bo około 40 osób uzyskało szansę poznania tego narzędzia.

Docker jest narzędziem które w łatwy sposób pozwala na tworzenie i dystrybucję aplikacji na różne środowiska. Dodatkowo umożliwia proste skalowanie i konfigurację naszych aplikacji. Pozwala w znaczący sposób skrócić proces tworzenia i testowania oprogramowania oraz natywnie korzysta z dwóch opisanych wcześniej funkcji jądra systemu, a mianowicie grup kontrolnych i przestrzeni nazw. Ważne jest również to, że Docker pozwala uruchomić niemal każdą aplikację bezpiecznie w kontenerze, dostarczając izolacjęi bezpieczeństwo na poziomie systemu. Pozwala to uruchamiać wiele kontenerów jednocześnie bez obaw, że będą one oddziaływały na siebie w jakikolwiek szkodliwy sposób. Wystarczy jedynie minimalny system operacji ze zgodną wersją jądra systemu oraz plik wykonalny Dockera.

W związku z tym, że aplikacje zostają uruchomione w izolowanym środowisku, Docker udostępnia narzędzia dla ułatwienia budowy, uruchamiania i zarządzania kontenerami. Oprócz możliwości uruchomienia go na lokalnym komputerze, istnieje również szereg dostawców usług, którzy wspierają kontenery Dockera, np. Google Cloud Platform, Microsoft Azure, Amazon EC2. Istnieje również platforma o nazwie Resin.io, która oferuje wsparcie systemów opartych na Dockerze dla urządzeń z zakresu Internetu Rzeczy, jednak dokładniej zapoznamy się z nią w rozdziale [link 2.8 Resin.io].

Cele Dockera

Głównymi celami Dockera sa:

- Dostarczenie łatwego sposobu modelowania złożonych systemów wymagających wielu współpracujących aplikacji. Dzięki użytym mechanizmom jest on bardzo szybki i łatwy do dostosowania do własnych potrzeb. Dodatkowo uruchamianie kontenerów aplikacji zajmuje w większości przypadków niespełna sekundę oraz nie wymaga uruchamiania nadzorców i zarządza naszymi zasobami systemowymi w bardziej optymalny sposób.

- Usprawnienie cyklu produkcji poprzez jego przyspieszenie i bardziej efektywne zarządzanie zasobami. Głównym celem jest tu redukcja czasu potrzebnego na tworzenie i testowanie oprogramowania, a zapewnione jest to głownie dzięki zapewnieniu możliwości uruchamiania tej samej aplikacji w wielu środowiskach bez dodatkowych zmian.

- Spójność aplikacji uruchamianych na różnych środowiskach, dzięki użyciu wirtualizacji opartej o kontenery. Zapewnia to, że nasza aplikacja będzie zachowywała się tak samo niezależnie od tego na jakim systemie uruchamiany jest kontener z nią.

Architektura Dockera

Docker bazuje na modelu klient-serwer. Klient Dockera komunikuje się z demonem[4], który odpowiada za tworzenie, uruchamianie oraz dystrybucję kontenerów. Zazwyczaj oba te komponenty uruchamiane są na tej samej maszynie, jednak możliwe jest uruchomienie demona na zdalnej maszynie i podłączenie się do niej za pomocą aplikacji klienckiej. Architektura została pokazana na poniższym rysunku. Każdy z komponentów ma określone zadania, które zostaną opisane poniżej.

Rysunek 2.6: Architektura wirtualizacji opartej o kontenery

Demon Dockera

Głównym zadanie tego komponentu jest zarządzanie działającymi kontenerami. Jak widać na powyższym rysunku demon uruchomiony jest na systemie, na którym działają kontenery. Klient natomiast używany jest w celu komunikacji z demonem, ponieważ użytkownik nie może bezpośrednio się z nim komunikować.

[4] [**?**]

Klient Dockera

Aplikacja kliencka Dockera jest, interfejsem uruchamianym z poziomu konsoli. Służy ona do komunikacji z procesem demona. Użytkownik poprzez odpowiednie komendy klienta może pośrednio komunikować się z demonem.

Obrazy Dockera

Obrazy są podstawowym źródłem służącym do tworzenia kontenerów. W pewnym sensie są one kodem źródłowym kontenerów. Obrazy można budować samemu praktycznie od zera lub użyć już istniejących obrazów dostępnych na platformach takich jak DockerHub[5]. Są one łatwe w konfiguracji i tworzeniu, a dzięki możliwości łatwego dzielenia się nimi, również łatwo dostępne.

Obrazy złożone są z wielu warstw instrukcji. Przykładowo, użytkownik może zdefiniować, aby przy tworzeniu obrazu uruchomiona została odpowiednia komenda systemowa, skopiowany folder czy ściągnięte dodatkowe zależności potrzebne do uruchomienia aplikacji. Następnie wszystkie warstwy są łączone w celu stworzenia obrazu. Dzięki takiemu podejściu do tworzenia obrazów, jeśli aktualizujemy tylko naszą aplikację, nie wymaga to przebudowy innych warstw, a jedynie aktualizacji warstwy zawierającej naszą aplikację.

Rejestr Dockera

Rejestry używane są do przechowywania stworzonych obrazów. Mogą być one prywatne lub publiczne. Publicznym rejestrem jest przykładowo wspomniany wcześniej DockerHub. Jest to jedna z najpopularniejszych platform służąca do przechowywania obrazów Dockera. Możemy z łatwością znaleźć obraz serwera dla naszej aplikacji webowej, bazę danych czy nawet cały system operacyjny. Jeśli nie chcemy natomiast dzielić się naszymi obrazami z innymi, możemy z łatwością skorzystać z prywatnych rejestrów, które również mogą być uruchamiane jako kontenery Dockera i udostępniać je tylko w obrębie naszej organizacji.

Kontenery

Jak już wcześniej wspomniano, kontenery tworzone są przy użyciu obrazów, które mogą zawierać również aplikacje i różne serwisy. Uruchamiają aplikacje w odizolowanym środowisku zawierającym wszelki zależności wymagane do ich działania.

[5] [**?**]

2.4 Systemy zarządzania kontenerami

Jedną z największych zalet kontenerów jest możliwość zarządzania nimi jako częścią klastra poprzez enkapsulację aplikacji w węzłach klastra. Załóżmy, że posiadamy pojedynczy serwer, na którym uruchomionych jest kilkaset kontenerów aplikacji, a każdy z nich wykonuje inne zadanie. Kontenery te mają dostęp do internetu i działają w sposób, który nie zakłóca pracy innych kontenerów na serwerze. Chcąc wykonać pewne operacje na większej grupie kontenerów, użytkownik może mieć trudności w zarządzaniu nimi. W takich sytuacjach z pomocą przychodzą systemy zarządzania kontenerami, które pozwalają nam grupować powiązane ze sobą kontenery aplikacji oraz zarządzać nimi w łatwy sposób poprzez swoje API[6]. Istnieje wiele systemów, jednak obecnie najpopularniejszymi z nich są Kubernetes (stworzony przez Google) oraz Docker Swarm (stworzony przez Dockera).

2.4.1 Kubernetes

Kubernetes jest otwartozródłowym systemem do zarządzania kontenerami aplikacji na wielu fizycznych i wirtualnych maszynach. Jego głównymi zaletami są: automatyczne rozlokowanie aplikacji w klastrze, autoskalowanie oraz zarządzanie istniejącymi aplikacjami. Został on stworzony w 2014 roku jako rezultat wielu lat doświadczeń firmy Google w zarządzaniu aplikacjami i kontenerami wewnątrz firmy. Dzięki Kubernetesowi możemy w łatwy sposób zainstalować naszą aplikację w klastrze, zaktualizować ją bez konieczności wyłączania czy zarządzać jej zasobami sprzętowymi w celu optymalizacji jej działania. Pozwala to na szybką reakcję w przypadku wystąpienia błędu lub wykorzystania limitu zasobów. Dodatkowo, Kubernetes jest przenośny, łatwy w uruchomieniu oraz może służyć jako publiczna, prywatna, a nawet hybrydowa chmura. Jego budowa oparta jest o moduły, które mogą być łatwo zastąpione, a każdy komponent ma swoje konkretne zadanie i zbudowany jest niezależnie od innych. Uznawany jest za system odporny na błędy, który zaprogramowany jest w taki sposób, aby radzić sobie w przypadku wystąpienia błędów, zazwyczaj bez konieczności ingerencji człowieka.

Kubernetes wprowadza szereg obiektów, które są abstrakcją mającą na celu wprowadzenie czytelnej i zrozumiałej dla użytkownika struktury systemu. Kontenery aplikacji użytkownika przedstawiane są jako pody [7], czyli podstawowa struktura systemu, grupująca kontenery i zarządzająca nimi. Przed ich uruchomieniem system bierze pod uwagę zasoby sprzętowe wszystkich węzłów klastra i decyduje, na którym z nich powinny zostać uruchomione kontenery. Każdej akcji towarzyszy

[6][]
[7][]

szereg decyzji, które muszą zostać podjęte przez kontrolery i planistę systemu w celu jak najefektywniejszego zarządzania zasobami.

Początki Kubernetesa - System Borg

Kubernetes jest następcą systemu zwanego Borg, który jest używany wewnętrzenie przez Google do zarządzania tysiącami zadań uruchomionych na setkach maszyn. Jarek Kuśmiarek prowadzący zespół deweloperów Borga w Warszawie opisuje go tak.

> *Podstawową jednostką w Borg jest centrum danych Google. Zaawansowany użytkownik wie, gdzie uruchomić swój program. Ten mniej zaawansowany pozostawia to systemowi Borg. Na decyzje podejmowane przez niego wpływają m.in. fakt, skąd dane są czytane najczęściej, kto z nich korzysta, a nawet to, w którym z naszych centrów danych planowane są naprawy serwisowe.*[8]

Podstawowymi funkcjami Borga są dystrybucja, uruchamianie i monitorowanie zadań oraz restart aplikacji. Dzięki zdjęciu z barków programistów konieczności zarządzania zasobami oraz błędami aplikacji pozwala on na skupienie się na samej aplikacji i jej funkcjonalności. Jako system wysokiej dostępności i niezawodności, może sam zarządzać i monitorować stan aplikacji.

Rysunek 2.7: Architektura systemu Borg

[8] http://itwiz.pl/borg-kubernetes-narzedzia-rozwijane-przez-google-polsce/[]

Powyżej została pokazana architektura omawianego systemu. Każda komórka składa się z wielu połączonych ze sobą maszyn oraz posiada kontroler główny *Borgmaster* i zespół agentów *Borglet*, które są węzłami na których uruchamiane są kontenery aplikacji. Wszystkie te komponenty zostały napisane w języku C++ w celu optymalizacji i jak najlepszej ich integracji z systemem. Borg zarządza pulą maszyn, a także tym, jakie systemy na nich pracują, ile zasobów potrzebują, oraz metodologią najbardziej efektywnego upakowania maszyn. Jak wspominają przedstawiciele Google, jeśli mamy mniej ważne i bardziej ważne zadania na tej samej maszynie, to możemy zarządzać nimi w sposób bardzo dynamiczny, zabierając i oddając ten sam zestaw zasobów z korzyścią dla systemu, który w danej chwili bardziej go potrzebuje. Borg zarządza też cyklem życia maszyn fizycznych. Informuje, gdy trzeba je naprawić. Wówczas znajdujące się na niej procesy automatycznie przenosi na inny serwer.

Opis poszczególnych komponentów, z których zbudowany jest Borg przedstawiony został poniżej:

Borgmaster: Główny komponent każdej komórki systemu Borg. Nasłuchuje on i przetwarza wszystkie żądania klientów i na ich podstawie tworzy nowe zadania lub umożliwia odczytanie danych. Zarządza on również stanem maszyn oraz umożliwia komunikację z Borgletami. Mimo iż ten komponent uruchamiany jest jako pojedyczny proces to jest on replikowany pięć razy w obrębie jednej komórki. Każdy proces posiada podpięty logiczny dysk bazujący na systemie Paxos19 w celu zachowania stanu systemu. Pozwala on na stworzenie systemu wysokiej dostępności, który zachowuje stan komórki w pamięci. Dodatkowo jeden z procesów jest wybierany jako lider, który jako centralny proces zarządza tworzeniem oraz kończeniem wszelkich zadań. Poprzez zastosowanie odpowiednich mechanizmów, nawet w przypadku błędu procesu lidera, system zachowuje działanie i w ciągu 10 sekund od braku kontaktu z liderem wybierany jest nowy lider.

Planista: Zajmuje się on optymalnym rozlokowaniem tworzonych zadań na dostępnych maszynach. Po zapisaniu każdego zadania jest ono dodawane do kolejki. Następnie zadania są asynchronicznie sprawdzane i przypisywane do maszyn na podstawie algorytmu Round-Robin[9], poprzez wybieranie zadań według priorytetów i zapewnienie sprawiedliwego rozlokowania zadań. Po wybraniu grupy maszyn, na których możliwe będzie wystartowanie danego zadania, są one poddawane procesowi oceniania. Na podstawie dostępnych zasobów, ilości uruchomionych zadań, a nawet

[9][]

zaplanowanych prac konserwacyjnych liczona jest ocena maszyn i wybierana ta z najlepszym wynikiem.

Borglet: Lokalny agent uruchomiony na każdej maszynie, której zadaniem jest uruchamianie zadań użytkowników. Zarządza i monitoruje wszystkie procesy na nim uruchomione i odpowiada za ich uruchamianie, zatrzymywanie czy restartowanie w razie potrzeby. Używa odpowiednich opcji jądra systemu w celu zarządzania zasobami oraz informowania głównego komponentu Borgmaster o dostępnych zasobach. Odpowiada na regularne zapytania ze strony Borgmastera, który weryfikuje stan węzła. Jeśli maszyna nie odpowie w wyznaczonym czasie, zostaje oflagowana, a wszystkie zadania na niej uruchomione zostają przeniesione na inne maszyny. W razie przywrócenia łączności wszelkie zduplikowane zadania zostają zatrzymane.

Architektura Kubernetesa

W celu konfiguracji Kubernetesa jako zarządcy kontenerów należy na maszynach zainstalować kilka serwisów/komponentów.

Rysunek 2.8: Architektura Kubernetesa

Jak widać na obrazku przedstawiającym architekturę systemu, komponenty są grupowane pod względem głównego węzła, na którym uruchomiony jest API serwer, kontroler główny, planista oraz istnieje możliwość uruchomienia procesu agenta w celu zarejestrowania węzła głównego również jako maszyny na której można uruchamiać aplikacje. Na pozostałych maszynach uruchomione zostają kolejno agent maszyny, czyli kubelet oraz proces proxy do zapewnienia odpowiedniej łączności z innymi maszynami. Dzięki takiemu podziałowi jesteśmy w stanie jeszcze lepiej

utylizować dostępne zasoby i zarządzać klastrem.

Węzeł główny(master) — ma za zadanie kontrolować klaster Kubernetesa. Jest to główny węzeł klastra, z którym komunikują się inne węzły jak i użytkownicy. Tylko on powinien być bezpośrednio udostępniony i widoczny dla użytkowników. Ma za zadanie uruchamianie, zarządzanie oraz monitorowanie aplikacji na wszystkich węzłach klastra(nodach), a osiąga to dzięki komunikacji z dodatkowymi komponentami opisanymi poniżej. W szczególności master również może być nodem, czyli węzłem na którym mogą być uruchamiane aplikacje użytkownika.

- *API serwer* — jest to kluczowy komponent całego klastra. Jako pośrednik między innymi komponentami, umożliwia użytkownikom konfigurację wszystkich ustawień klastra, od autentykacji, poprzez tworzenie aplikacji, aż po zarządzanie regułami sieciowymi. Pozwala na dostęp do aplikacji uruchomionych na innych węzłach klastra dzięki regułom sieciowym tworzonym przez inny komponent zwany *kube-proxy*.

- *Kontroler główny* — jest to demon uruchomiony na masterze w celu obsługi zadań zleconych przez API serwer. Główna pętla aplikacji obserwuje zmiany zachodzące w klastrze. Kiedy taka zmiana następuje, kontroler odczytuje nowy stan obiektu i aktualizuje istniejący obiekt do nowego stanu.

- *Planista* — zarządza on wszystkimi zadaniami i aplikacjami w trakcie ich tworzenia poprzez podejmowanie decyzji, na którym węźle klastra powinny zostać uruchomione. Odpowiednie algorytmy oceniania biorą pod uwagę wiele czynników, takich jak: aktualne obciążenie klastra, dostępne zasoby, planowane prace konserwacyjne i wiele więcej. W razie gdy jeden z węzłów przestanie odpowiadać jego zadaniem jest relokacja aplikacji na nowy węzeł w celu zapewnienia dostępności aplikacji.

- *Etcd*: jest to niezawodny i szybki magazyn do przechowywania danych w postaci klucz - wartość. Został stworzony przez firmę CoreOS. Jego głównymi zaletami są: proste API, bezpieczeństwo, szybkość oraz niezawodność. Został napisany w języku Go w oparciu o algorytm Raft [10].

Węzeł klastra(node) — każdy węzeł klastra wymaga kilku komponentów, które są niezbędne do zapewnienia komunikacji z węzłem głównym i uruchamiania aplikacji. W celu zapewnienia odpowiedniej komunikacji między aplikacjami na różnych węzłach, każdemu z nich przydzielana jest odpowiednia dedykowana podsieć. Poszczególne komponenty zostały opisane poniżej.

[10][]

- *Docker* — pierwszym i najważniejszym wymogiem jest uruchomiony serwis Dockera na każdym węźle klastra, który ma uruchamiać aplikacje. Jest on używany w celu tworzenia aplikacji opartych o kontenery.

- *Kubelet* — ten komponent jest używany w celu zarządzania aplikacjami uruchomionymi na danym węźle. W szczególności zarządza on abstrakcyjnymi obiektami stworzonymi przez Kubernetesa zwanymi podami w celu uproszczenia procesu konfiguracji aplikacji. Funkcjonuje on jako agent, który umożliwia poprzez komunikację z API serwerem na zapisywanie stanu obiektów w magazynie danych etcd. Może on działać niezależnie od głównego węzła, w razie utracenia połączenia, a po jego przywróceniu zaktualizować stan aplikacji w oparciu o informacje uzyskane z mastera. Po pierwszym uruchomieniu rejestruje on węzeł w klastrze, a następnie raportuje zużycie zasobów do mastera, dzięki czemu planista może odpowiednio ocenić dany węzeł w trakcie procesu oceniania.

- *Proxy* — ten komponent uruchomiony jest na każdym węźle klastra i umożliwia ich udostępnienie światu. Jest on odpowiedzialny ze tworzenie reguł w tablicy routingu i przekierowywanie żądań do odpowiednich kontenerów aplikacji.

Dedykowane obiekty Kubernetesa: Kubernetes tworzy kontenery aplikacji w oparciu o stworzone przez siebie abstrakcyjne obiekty, które mają za zadanie oddzielić zarządzanie aplikacją, tj.: konfigurację sieci w celu udostępnienia aplikacji światu, skalowanie aplikacji czy restartowanie w razie wystąpienia błędów. Poniżej zostały opisane najczęściej używane obiekty. Warto zaznaczyć, że jest ich dużo więcej.

- *Pod* — jest podstawowym obiektem tworzonym przez Kubernetesa, który zarządza kontenerami aplikacji. Umożliwia on użytkownikom grupowanie powiązanych ze sobą kontenerów w jeden obiekt. Zamiast uruchamiać wiele zależnych kontenerów wielu Podach możemy uruchomić wiele aplikacji w obrębie jednego Poda, a Kubernetes będzie nim zarządzać jak jedną aplikacją. Pozwala to na uruchamianie wszystkich zależnych aplikacji w obrębie tego samego węzła klastra, a co za tym idzie współdzielenie zmiennych środowiskowych czy zamontowanych dysków. Dodatkowo kontenery w obrębie jednego Poda posiadają wspólny adres IP. Zazwyczaj Pod zawiera kontener główny, który odpowiada za udostępnianie tzw. frontendu[11] użytkownikowi oraz kontener będący backendem[12] dla aplikacji głównej.

[11] [**?**]
[12] [**?**]

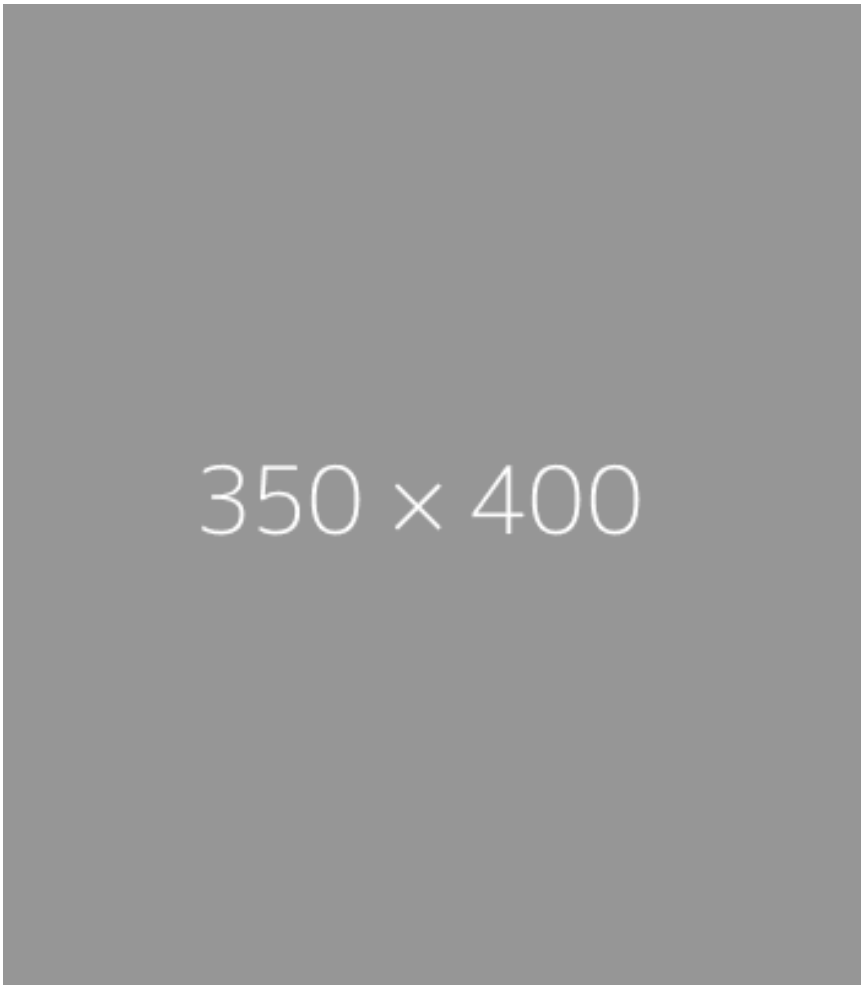
- *Replication controller* — jest on przede wszystkim zarządcą grupy powiązanych Podów. Jego rolą jest zapewnienie aby zdefiniowana liczba Podów była zawsze uruchomiona w klastrze. Przykładowo, w razie wykrycia zbyt dużej ilości powiązanych Podów usunie nadmiarowe. W odwrotnej sytuacji, jeśli liczba Podów będzie zbyt niska w stosunku do zdefiniowanej, utworzy on nowe. Można go określić jako nadzorcę, który pozwala na łatwe zarządzanie grupą Podów uruchomionych w obrębie klastra.

- *Services* — serwisy są abstrakcyjnymi obiektami, które definiują politykę dostępu do grupy Podów definiujących tą samą aplikację, a posiadających różne adresy IP. Zazwyczaj w celu rozróżnienia, które Pody podlegają pod dany serwis używane są etykiety[13] definiowane przy tworzeniu Podów czy Replication Controllera. W związku z tym, że Pody mogą być usuwane i odtwarzane, co zazwyczaj wiąże się ze zmiana adresu IP, stworzone zostały właśnie serwisy. Dzięki serwisom użytkownik zyskuje dostęp do swojej aplikacji mimo, że może być ona uruchomiona na wielu różnych węzłach klastra. Możemy wyróżnić kilka rodzajów serwisów, które odpowiadają za udostępnianie aplikacji wewnątrz klastra, w obrębie węzła oraz jako Load Balancer[14] w celu udostępnienia aplikacji na świat oraz odpowiedniego kierowania ruchem do naszej aplikacji.

2.4.2 Docker Swarm

Docker Swarm umożliwia tworzenie klastra w oparciu o kontenery Dockera, jednak jego zaletą jest głęboka integracja z samym Dockerem, dzięki czemu jest on natywnie wspierany. Nie jest on tak zaawansowany jak Kubernetes jednak wystarczający do prostych zastosowań. Posiada możliwość grupowania kontenerów w grupy oraz zarządzania nimi w prosty sposób, podobny do zarządzania samymi kontenerami Dockera. Sama architektura jak widać na poniższym obrazku jest dużo prostsza niż w przypadku Kubernetesa. Posiada on planistę oraz Discovery Service, którego zadaniem jest udostępnianie aplikacji oraz umożliwienie komunikacji między aplikacjami wewnątrz klastra, a które uruchomione są na różnych węzłach.

[13] [**?**]
[14] [**?**]



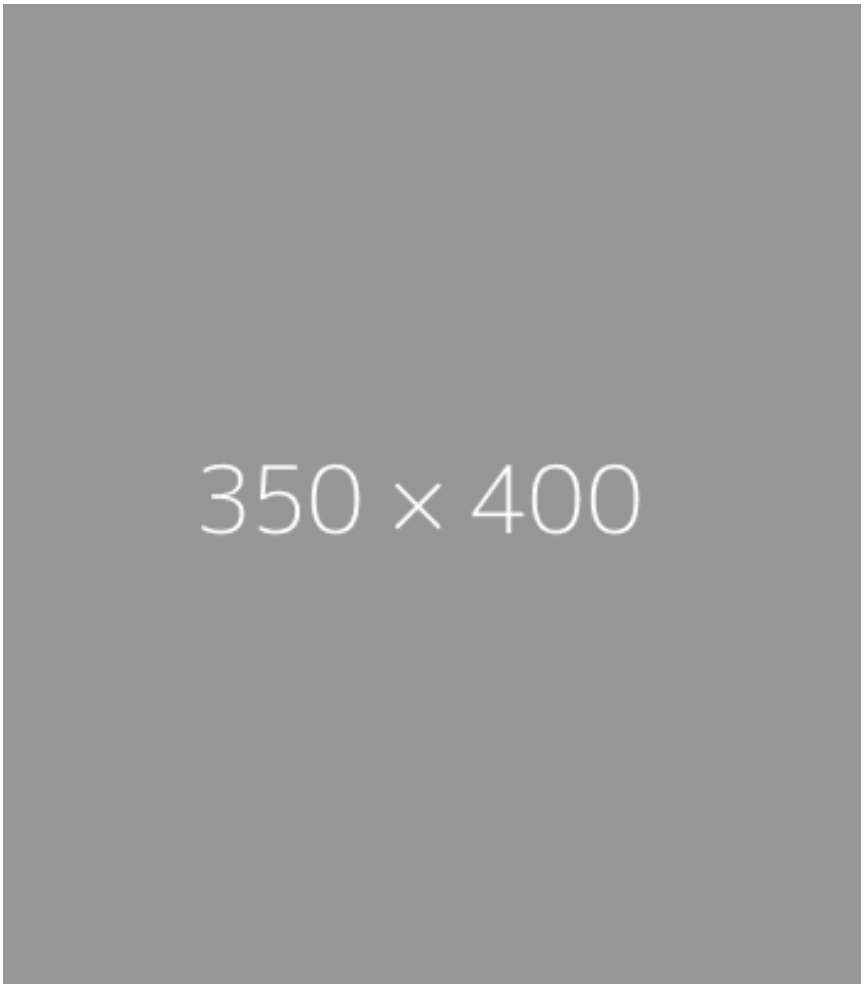
Rysunek 2.9: Architektura Docker Swarm

2.4.3 Resin.io

Resin jest rozwiązaniem przygotowanym w celu przeniesienia zalet kontenerów na mikrokontrolery. Obecnie jako jedyny, umożliwia zdalne uruchamianie kontenerów i zarządzanie grupą mikrokontrolerów ze wspólnego panelu sterowania. Dużą zaletą jest wsparcie wielu różnych urządzeń, od RaspberryPi, aż po Intel NUC. To wszystko pozwala deweloperom skupić się na dostarczeniu samej aplikacji oraz sprzętu, na którym będziemy mogli uruchomić naszą aplikację. Pozostałe kroki wymagane do uruchomienia aplikacji na urządzeniu są zarządzane przez system Resin.io, a są to między innymi:

- Przeniesienie kodu na urządzenie.

- Zdalna kompilacja kodu z uwzględnieniem odpowiedniej architektury urządzenia.
- Monitorowanie, zarządzanie i kontrola aplikacji na naszych urządzeniach.
- Zapewnienie bezpieczeństwa komunikacji.
- Dostarczenie środowiska uruchomieniowego dla aplikacji.



Rysunek 2.10: Architektura Resina

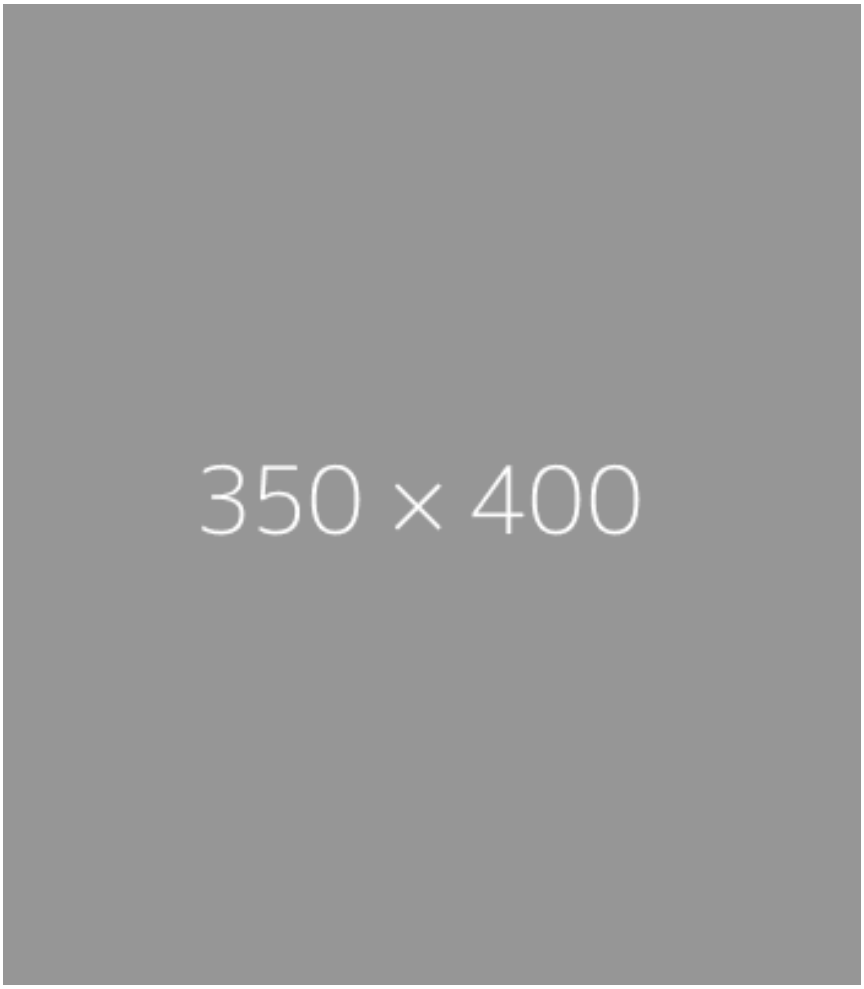
Jednak pomimo wielu zalet, nie jest on na tyle zaawansowany i bezpieczny w porównaniu do Kubernetesa. Głównym ograniczeniem jest możliwość uruchomienia jedynie jednego kontenera aplikacji na każdym urządzeniu użytkownika.

Grupowanie urządzeń oraz definiowanie zaawansowanych reguł sieciowych w celu ograniczenia komunikacji między urządzeniami również nie jest możliwe.

2.5 Mikrokontroler Raspberry Pi

Stworzona przez Raspberry Pi Foundation seria mikrokontrolerów Raspberry Pi miała swój początek w Wielkiej Brytanii, gdzie promowane były one jako idealne do nauki podstaw budowy i działania komputerów. Przez lata kolejne modele były ulepszane i obecnie są one używane w zagadnieniach Internetu Rzeczy, robotyce, inteligentnych domach i wielu innych dziedzinach.

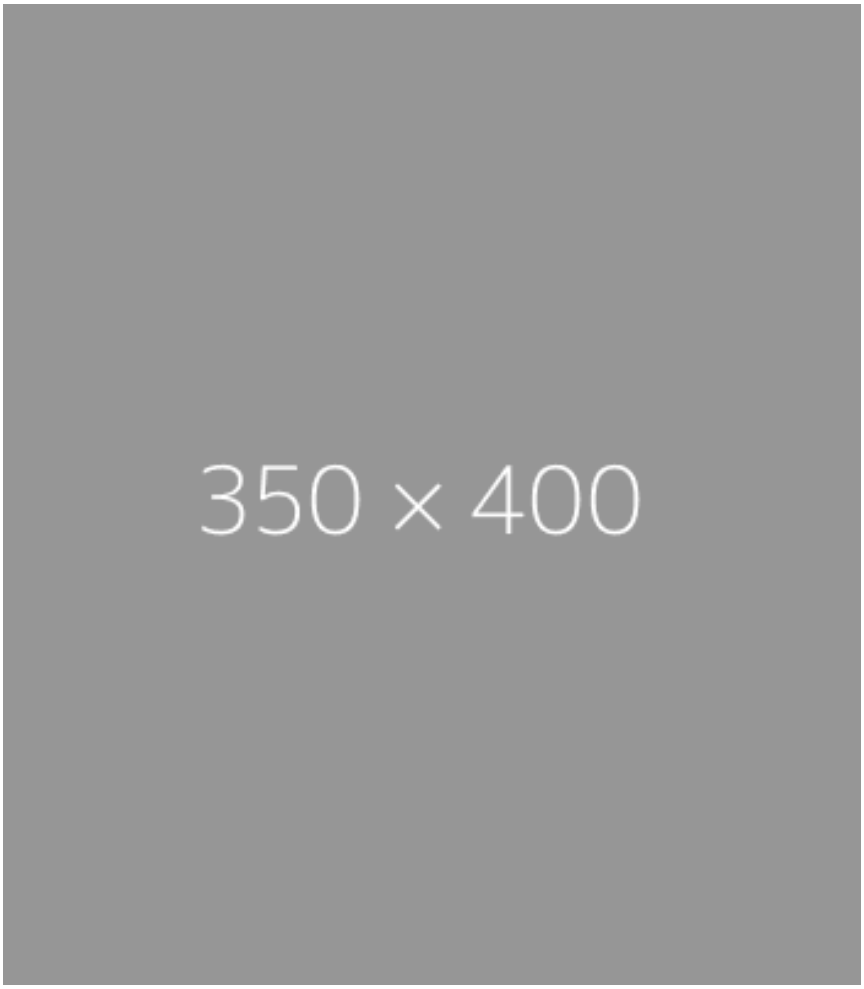
Poniższa tabela prezentuje porównanie najpopularniejszych modeli Raspberry Pi, tj.: A, B, A+, B+, wersja druga modelu B oraz 3. W pracy zostaną użyte modele wersja druga B oraz najnowsza obecnie Raspberry Pi 3, w której znaczącej poprawie uległy taktowanie procesora oraz wielkość pamięci RAM. Dzięki temu może być ona stosowana jako główny węzeł klastra, kontrolujący pozostałe węzły.



Rysunek 2.11: Mikrokontroler Raspberry Pi

2.5.1 Protokoły komunikacji

Poniżej opisane zostały protokoły komunikacji Raspberry Pi, użyte na potrzeby wykonania tej pracy. Protokół SPI użyty został do podłączenia wyświetlacza LED i wyświetlania aktualnej temperatury raportowanej przez czujniki. Jeden z czujników komunikuje się poprzez protokół 1-Wire, natomiast drugi komunikuje się bezpośrednio poprzez piny GPIO. Ze względu na to, że kontrolery Raspberry Pi B+ oraz 3 posiadają identyczny układ pinów, poniżej pokazany został jedynie układ pinów Raspberry Pi 3.



Rysunek 2.12: Układ pinów mikrokontrolera Raspberry Pi 3

Serial Peripheral Interface - SPI

Synchroniczny interfejs komunikacji używany głównie przy komunikacji na krótkie dystanse, przeważnie w systemach wbudowanych i mikrokontrolerach. Typowo stosowany do bezpiecznych kart cyfrowych oraz wyświetlaczy LED.

Urządzenia komunikujące się przy użyciu interfejsu SPI działają w trybie full duplex, czyli jednoczesnej komunikacji dwukierunkowej. Nadrzędne urządzenie inicjuje komunikację i umożliwia odczyt oraz zapis danych. Dodatkowo możliwa jest komunikacja z wieloma urządzeniami podrzędnymi poprzez wybór odpowiedniej linii sygnałowej.

1-Wire

Prosty protokół komunikacyjny, którego nazwa wywodzi się z faktu użycia pojedynczej linii danych do komunikacji. Mimo niskiej prędkości przesyłu danych charakteryzuje się dużą popularnością. Wspiera tzw. zasilanie pasożytnicze, a mianowicie pozwala na zasilanie czujnika bezpośrednio z linii danych. Odbiornik wyposażony w kondensator jest ładowany z linii danych, a następnie ta energia używana jest do zasilania odbiornika. Używany najczęściej w przypadku prostych czujników, np. do raportowania temperatury i wilgotności otoczenia.

General Purpose Input/Onput - GPIO

Interfejs ten służy do komunikacji między różnymi elementami systemu, takimi jak mikroprocesor czy urządzenia peryferyjne. Fizycznie komunikacja odbywa się przy użyciu pinów wbudowanych w dane urządzenie, które mogą pełnić rolę wejść i wyjść w zależności od konfiguracji. Często grupowane są one w porty , gdzie jeden port stanowi zazwyczaj osiem pinów GPIO.

Komunikacja odbywa się poprzez ustawienie stanu wysokiego lub niskiego na odpowiednim pinie GPIO. Z tego względu pojedynczy pin może służyć jedynie do prostych czynności takich jak wykrywanie zdarzeń, gazów. Natomiast sterowanie grupą pinów jako portem daje dużo większe możliwości.

Rozdział 3

Projekt KubePi

3.1 Analiza wymagań

W tej części zostanie przedstawiona analiza, konfiguracja oraz implementacja systemu KubePi. W pierwszej części zdefiniowane zostają wymagania aplikacji oraz opisane wymagania funkcjonalne wraz z ograniczeniami projektu. Kolejna część opisuje technologie użyte przy tworzeniu projektu. Następnie przedstawiona zostaje propozycja i architektura systemu wraz z opisem implementacji przykładowych aplikacji. Przybliżone zostają kluczowe elementy konfiguracji systemu wraz z opisem kodu najważniejszych elementów aplikacji.

3.1.1 Studium możliwości

Główną ideą projektu jest rozwiązanie problemu stworzenia rozproszonego systemu wysokiej dostępności opartego na architekturze ARM i mikrokontrolerach. Zaletami takiego rozwiązania jest obniżenie kosztów budowy klastra kosztem wydajności. Jednak ze względu na charakter przedsięwzięcia i użycie mikrokontrolerów Raspberry Pi do projektu monitorowania otoczenia, możemy pozwolić sobie na takie ustępstwa, ponieważ wydajność nie jest kluczowa, a stabilność systemu przy odpowiedniej architekturze pozostaje na tym samym poziomie. Dodatkowo po przygotowaniu takiego systemu używanie, zarządzanie i konfiguracja stają się bardzo proste.

3.1.2 Wymagania funkcjonalne

Ze względu na swój charakter oraz wymóg komunikacji między aplikacjami znajdującymi się na różnych węzłach klastra system musi spełniać następujące wymagania:

1. Możliwość komunikacji między węzłami klastra.
2. Możliwość komunikacji między aplikacjami znajdującymi się na tych samych i różnych węzłach klastra.
3. Zarządzanie klastrem i aplikacjami powinno być proste i przejrzyste.
4. System powinien zachowywać swój stan w przypadku niezamierzonego restartu systemu.
5. W przypadku restartu system powinien samoczynnie wrócić do stanu z przed restartu bez ingerencji użytkownika.

Dodatkowo system powinien umożliwiać bezpieczny sposób komunikacji z klastrem oraz autentykację i autoryzację użytkowników. Ze względu na obecny poziom skomplikowania pracy ten element zostanie pominięty, jednak system będzie umożliwiał odpowiednią konfigurację zabezpieczeń klastra.

3.1.3 Ograniczenia projektu

Ze względu na skomplikowany proces konfiguracji systemu i klastra, projekt nie będzie umożliwiał łatwego przeniesienia i uruchomienia w innej podsieci. Wymóg stosowania odpowiedniej topologii i narzędzi do zarządzania siecią w celu umożliwienia odpowiedniej komunikacji między elementami systemu wymaga aby urządzenia miały przydzielany stały adres IP lub minimalnie adres z wcześniej skonfigurowanego zakresu podsieci. Ta opcja jednak wymagałaby dodatkowych kroków i odpowiedniej konfiguracji serwera DNS[1], co nie zostało uwzględnione w tym projekcie.

3.2 Użyte technologie

3.2.1 Język programowania

Do stworzenia aplikacji przykładowych wybrany został język Go. Najważniejszym czynnikiem decydującym za tym wyborem była obecność dużej ilości bibliotek ułatwiających komunikację, ze sprzętem oraz łatwa kompilacja aplikacji przeznaczonych na różne architektury, w tym ARM. Dodatkowym czynnikiem decydującym za językiem Go jest domyślna kompilacja aplikacji do statycznego pliku binarnego, który nie wymaga dodatkowych zależności, a co za tym idzie w prosty sposób

[1] [**?**]

może zostać uruchomiony w kontenerze Dockera. Ostatnią zaletą jest mała zajętość pamięci i czasu procesora w porównaniu z językami takimi jak Java, co jest bardzo ważne przy uruchamianiu aplikacji w kontenerze na mikrokontrolerach typu Raspberry Pi.

Dodatkowo w celu stworzenia aplikacji webowej użyta została biblioteka ReactJS. Głównym powodem przemawiającym za wyborem tej biblioteki była popularność, duże wsparcie ze strony społeczeństwa oraz niewielki rozmiar i prostota użycia.

Rysunek 3.1: Logo języka Go i biblioteki ReactJS

3.2.2 Biblioteki

- go-restful
- go-rpio
- go-dht
- goDS18B20
- spidev

3.2.3 Inne narzędzia

Flannel

Github

Kubernetes Dashboard

3.3 Projekt

3.3.1 Przygotowanie i konfiguracja Raspberry Pi

3.3.2 Instalacja i konfiguracja Kubernetesa

3.3.3 Projekt i konfiguracja aplikacji przykładowych

3.4 Opis użytkowania

3.5 Przykład użycia

3.6 Możliwości rozszerzania aplikacji

Rozdział 4

Podsumowanie

Bibliografia

[1] TODO

Spis rysunków

Spis tablic