

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

<b>1</b>	<b>Wstęp</b>	<b>4</b>
1.1	Problematyka i zakres pracy . . . . .	4
1.2	Uzasadnienie wyboru tematu . . . . .	5
1.3	Metoda badawcza . . . . .	5
1.4	Przegląd literatury w dziedzinie . . . . .	7
1.5	Układ pracy . . . . .	8
<b>2</b>	<b>Podstawy teoretyczne</b>	<b>9</b>
2.1	Internet Rzeczy . . . . .	9
2.1.1	Obszary zastosowań . . . . .	10
2.2	Chmury Obliczeniowe . . . . .	11
2.2.1	Architektura . . . . .	12
2.3	Wirtualizacja . . . . .	13
2.3.1	Wirtualizacja oparta o nadzorcę . . . . .	14
2.3.2	Wirtualizacja oparta o kontenery . . . . .	15
2.3.3	Docker . . . . .	19
2.4	Systemy zarządzania kontenerami . . . . .	22
2.4.1	Kubernetes . . . . .	22
2.4.2	Docker Swarm . . . . .	28
2.4.3	Resin.io . . . . .	29
2.5	Mikrokontroler Raspberry Pi . . . . .	31
2.5.1	Protokoły komunikacji . . . . .	32
<b>3</b>	<b>Projekt KubePi</b>	<b>35</b>
3.1	Analiza wymagań . . . . .	35
3.1.1	Studium możliwości . . . . .	35
3.1.2	Wymagania funkcjonalne . . . . .	35
3.1.3	Ograniczenia projektu . . . . .	36
3.2	Sprzęt i technologie . . . . .	36

3.2.1	Mikrokontrolery . . . . .	36
3.2.2	Urządzenia peryferyjne . . . . .	37
3.2.3	Języki programowania . . . . .	39
3.2.4	Biblioteki . . . . .	39
3.2.5	Inne narzędzia . . . . .	40
3.3	Projekt . . . . .	43
3.3.1	Architektura systemu . . . . .	43
3.3.2	Przygotowanie i konfiguracja Raspberry Pi . . . . .	47
3.3.3	Instalacja i konfiguracja Kubernetesa . . . . .	50
3.3.4	Projekt aplikacji przykładowych . . . . .	54
3.4	Opis użycia . . . . .	69
3.5	Możliwości rozszerzania aplikacji . . . . .	69
<b>4</b>	<b>Podsumowanie</b>	<b>70</b>
	<b>Bibliografia</b>	<b>70</b>
	<b>Spis rysunków</b>	<b>71</b>
	<b>Spis tabel</b>	<b>72</b>

# 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 przysyłać 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 <sup>1</sup> 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-

---

<sup>1</sup>[?]

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 virtualizacji opartej o kontenery Dockera<sup>2</sup> zarządzane przez system zarządzania klastrem Kubernetes i ma na celu stworzenie rozproszonego systemu służącego do monitorowania otoczenia. Przykładowy klastrowy 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<sup>3</sup>, co pozwoli na odczyt i wyświetlanie temperatury raportowanej przez pierwsze urządzenie. Dodatkowo w klastrze zostanie zainstalowana aplikacja webowa<sup>4</sup> 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.

## 1.3 Metoda badawcza

1. Studia literaturowe z dziedziny generowania opisu mapowania obiektowo-relacyjnego w języku C++.

Obecnie dostępne źródła z tej dziedziny nie są sformalizowane. Dostępne są jedynie opisy i dokumentacje istniejących aplikacji, które uwzględniają

---

<sup>2</sup>[?]

<sup>3</sup>[?]

<sup>4</sup>[?]

w sposób ogólny ich budowę. Znalezione i użyte w tej pracy źródła nie są dostępne w języku polskim, więc muszą być tłumaczone w większości z języka angielskiego. Jako że nie ma oficjalnych książek dotyczących tematyki generowania opisu mapowania obiektowo-relacyjnego, większość źródeł tu zebranych to źródła elektroniczne, artykuły i dokumentacje.

2. Analiza wymagań aplikacji szkieletowych generujących opis mapowania obiektowo-relacyjnego.

Narzędzia zajmujące się generowaniem opisu mapowania obiektowo-relacyjnego są zazwyczaj tylko dodatkami do typowych aplikacji typu ORM. Nie znajdziemy tu modelu aplikacji, na którym można bazować. Wymagania postawione takiej aplikacji są zazwyczaj takie same i są one podyktowane przez aplikacje zajmujące się mapowaniem obiektowo-relacyjnym. Podobnie jest i w tym przypadku gdzie wymagania generatora opisu są postawione przez drugi moduł aplikacji zajmujący się mapowaniem obiektowo-relacyjnym.

3. Proces projektowania i tworzenia Qubica.

W oparciu o zebrane informacje i wymagania aplikacji szkieletowych służących do generowania opisu mapowania obiektowo-relacyjnego zostanie stworzona aplikacja szkieletowa mająca na celu rozwiązanie problemów zidentyfikowanych w procesie analizy.

4. Testy i wnioski dotyczące stworzonego narzędzia do generowania warstwy danych aplikacji w oparciu o bazę danych.

Metoda ta służy do wyciągnięcia wniosków na temat stworzonej aplikacji. Przeprowadzone zostaną testy porównawcze. Na podstawie wyników testów wyciągnięte zostaną odpowiednie wnioski na temat sposobu rozwiązania przedstawionych w pracy problemów oraz Qubica. Wszystko to pozwoli stwierdzić czy proponowane rozwiązanie jest lepsze, tańsze, szybsze od porównywanych.

## 1.4 Przegląd literatury w dziedzinie

### Źródła z zakresu języka C++

Użyte w tej pracy źródła dotyczące języka C++ służą przede wszystkim poznaniu technik programowania bibliotek współdzielonych oraz technik metaprogramowania. Dodatkowym celem przy pisaniu samej aplikacji jest chęć poznania nowego standardu języka C++11, który również jest przedstawiony w użytych źródłach. Szczegółowe omówienie tego standardu zostało przedstawione na stronie twórcy języka i służyć będzie jako główne źródło wiedzy [?]. Sposób tworzenia bibliotek i techniki metaprogramowania zostały opisane w książce *Advanced C++ Metaprogramming* [?].

### Źródła z zakresu narzędzi i aplikacji do mapowania obiektowo-relacyjnego

Tematyka generowania opisu mapowania obiektowo-relacyjnego jest związana z narzędziami ORM i brak jest książek dedykowanych tej tematyce. Do zrozumienia samej idei działania generatora należy przybliżyć działanie narzędzi do mapowania obiektowo-relacyjnego. W pozycjach *EJB 3 Java persistence API* [?] oraz *Hibernate w akcji* [?] znajdziemy opis działania narzędzi typu ORM oraz techniki mapowania obiektowo-relacyjnego.

### Źródła z zakresu działania aplikacji szkieletowej Qt

Aplikacja szkieletowa Qt to zestaw bibliotek i narzędzi przydatnych programistom. Dzięki mechanizmowi refleksji, wsparciu dialektów SQL czy prostej budowie aplikacji graficznych znacznie ułatwia tworzenie dużych aplikacji. Użyta książka *Introduction to Design Patterns in C++ with Qt* [?] opisuje w prosty sposób mechanizm refleksji, wzorce czy tworzenie bibliotek przy użyciu tego frameworka. Dodatkowo głównym narzędziem w etapie tworzenia aplikacji będzie dokumentacja Qt dostępna w internecie pod adresem [?].

### Źródła z zakresu SQL

W celu generowania opisu bazy danych potrzebna jest znajomość struktury bazy, typów pól, połączeń. Wymaga to dla niektórych dialektów SQL pisania dość nietypowych zapytań. Potrzebne informacje zostały zasięgnięte ze źródeł elektronicznych i odpowiednich dokumentacji konkretnych dialektów, m. in: stron internetowa z dokumentacją dialektu MySQL [?].

## 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 krótko przedstawione zostają tematyki związane z mikrokontrolerami oraz Internetem Rzeczy.

Kolejny rozdział opisuje 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 zebranych 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 wcześniej 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 użycia 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. Infrastructure-as-a-Service) — znana również pod nazwą hardware-as-a-service<sup>1</sup>. Jest najprostszą odmianą usługi chmurowej 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<sup>2</sup> oraz Google App Engine<sup>3</sup>
- Oprogramowanie jako serwis (z ang. Software-as-a-Service) — ostatni z modeli usług chmurowej 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.

---

<sup>1</sup>[?]

<sup>2</sup>[?]

<sup>3</sup>[?]



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 (goście). Pełni on rolę emulatora i izoluje wirtualne systemy operacyjne od sprzętu jak i od hosta, czyli działa dwa poziomy ponad sprzętem. Popularnymi przykładami 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 uruchamiania 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 izolację 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 zamontowanie systemu 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 daną 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 książkę], 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 izolacji 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 są:

- 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 głównie 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 demodem<sup>4</sup>, 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 połączenie 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 zadaniem 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.

---

<sup>4</sup>[?]

### **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<sup>5</sup>. 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 wszelkie zależności wymagane do ich działania.

---

<sup>5</sup>[?]

## 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<sup>6</sup>. 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 otwartoźródłowym 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<sup>7</sup>, 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

---

<sup>6</sup>□

<sup>7</sup>□

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 wewnętrznie 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.<sup>8</sup>*

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

---

<sup>8</sup><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 pojedynczy proces to jest on replikowany pięć razy w obrębie jednej komórki. Każdy proces posiada podpięty logiczny dysk bazujący na systemie Paxos<sup>19</sup> 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<sup>9</sup>, 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 oceny. Na podstawie dostępnych zasobów, ilości uruchomionych zadań, a nawet

---

<sup>9</sup>□



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 server, 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

użyłować dostępne zasoby i zarządzać klastrem.

**Węzeł główny(master)** — ma za zadanie kontrolować klastr 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 server* — 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 server. 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 węzle 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 <sup>10</sup>.

**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. Po szczególne komponenty zostały opisane poniżej.

---

<sup>10</sup>[]

- *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 za 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<sup>11</sup> użytkownikowi oraz kontener będący backendem<sup>12</sup> dla aplikacji głównej.

---

<sup>11</sup>[?]

<sup>12</sup>[?]

- *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<sup>13</sup> definiowane przy tworzeniu Podów czy Replication Controllera. W związku z tym, że Pody mogą być usuwane i odtwierzane, co zazwyczaj wiąże się ze zmianą 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<sup>14</sup> 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.

---

<sup>13</sup>[?]

<sup>14</sup>[?]



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.



Rysunek 2.11: Mikrokontroler Raspberry Pi

Poniższa tabela prezentuje porównanie najpopularniejszych modeli Raspberry Pi, tj.: A, B, A+, B+, wersji drugiej modelu B oraz wersji trzeciej. W pracy zostały użyte dwa najnowsze obecnie Raspberry Pi 3, w których znaczącej poprawie uległy taktowanie procesora oraz wielkość pamięci RAM. Dzięki temu mogą być one stosowane jako główny węzeł klastra, kontrolujący pozostałe węzły.

### **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/Output - 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<sup>1</sup>, co nie zostało uwzględnione w tym projekcie.

## 3.2 Sprzęt i technologie

### 3.2.1 Mikrokontrolery

Na potrzeby projektu i budowy klastra zostały użyte dwa mikrokontrolery Raspberry Pi, jeden w wersji drugiej, model B, natomiast drugi w wersji trzeciej. Na obu mikrokontrolerach zainstalowany został darmowy system HypriotOS <sup>2</sup>, który natywnie wspiera Dockera i jest dobrze przygotowany do obsługi Kubernetesa. Nowszy model płytki w wersji trzeciej, ze względu na lepsze parametry techniczne

---

<sup>1</sup>[?]

<sup>2</sup>HypriotOS - link

został użyty jako główny węzeł klastra, a zarazem jednostka sterująca i zarządzająca klastrem. Druga płytki natomiast została użyta jako dodatkowy węzeł klastra. Ilość węzłów klastra może być bez problemu zwiększona. Kubernetes w najnowszej obecnie wersji, tj. 1.6 może obsłużyć nawet do 5000 węzłów.

### 3.2.2 Urządzenia peryferyjne

#### DHT11

Popularny czujnik temperatury i wilgotności powietrza z interfejsem cyfrowym, jednoprzewodowym. Zakres pomiarowy: temperatura 0 °C - 50 °C , wilgotność 20 - 90 %RH [?].



Rysunek 3.1: Czujnik temperatury i wilgotności DHT11.

#### DS18B20

Cyfrowy czujnik temperatury DS18B20 z interfejsem 1-wire. Działa w zakresie od -55 °C do 125 °C. Zasilany jest napięciem od 3,0 V do 5,5 V [?].



Rysunek 3.2: Czujnik temperatury DS18B20.

**MAX7219**

Moduł matrycy złożonej z 128 czerwonych diod LED ułożonych w prostokąt 16 x 8 pikseli. Płytkę posiada zamontowane złącza dopasowane do pinów GPIO Raspberry Pi [?].



Rysunek 3.3: Wyświetlacz LED MAX7219 (16x8).

### 3.2.3 Języki 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 architekturę ARM. Dodatkowym czynnikiem decydującym o wyborze języka Go jest domyślna kompilacja aplikacji do statycznego pliku binarnego, który nie wymaga dodatkowych zależności, a co za tym idzie w prosty sposób 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 atutem przemawiającym za wyborem tej biblioteki była popularność, duże wsparcie ze strony społeczności oraz niewielki rozmiar i prostota użycia.



Rysunek 3.4: Logo języka Go i biblioteki ReactJS

### 3.2.4 Biblioteki

Do stworzenia aplikacji przykładowych użytych zostało kilka otwartoźródłowych bibliotek dostępnych na portalu Github. Poniżej pokrótce została opisana każda z nich:

- *go-restful* — Jedna z najpopularniejszych bibliotek dla języka Go używanych do tworzenia tak zwanych RESTful Webservices<sup>3</sup>.
- *go-rpio* — Mała biblioteka pozwalająca w prosty sposób na obsługę pinów GPIO mikrokontrolerów Raspberry Pi. Użyta przy tworzeniu aplikacji do wykrywania alkoholu.
- *go-dht* — Biblioteka wspierająca czujniki temperatury i wilgotności DHT11 i DHT22. Pozwala na odczyt pomiarów czujnika na mikrokontrolerach Raspberry Pi.
- *goDS18B20* — Podobnie do biblioteki *go-dht* pozwala na odczyt pomiarów czujnika temperatury *DS18B20*.
- *spidev* — Biblioteka pozwalająca na obsługę urządzeń wykorzystujących interfejs komunikacyjny SPI. Użyta do obsługi wyświetlacza led *MAX7219*.
- *ReactJS* — Biblioteka stworzona przez firmę Facebook napisana w języku JavaScript, pozwalająca w prosty sposób tworzyć interfejs graficzny aplikacji internetowych.

### 3.2.5 Inne narzędzia

#### Flannel

Jest menadżerem wirtualnej sieci, który umożliwia stworzenie wirtualnej podsieci dla każdego węzła w klastrze. W dużym uproszczeniu pozwala on na komunikację między aplikacjami znajdującymi się w kontenerach uruchomionych na różnych węzłach klastra, przy czym nie jest wymagane aby węzły klastra przebywały w tej samej podsieci. Dzięki takim narzędziom jak Flannel, zwanym *Overlay Networks* możliwa jest komunikacja między aplikacjami uruchomionymi w rozproszonym klastrze.

#### Github

Dodatkowym narzędziem użytym w procesie tworzenia aplikacji był system kontroli wersji o nazwie GIT. Pozwalało to kontrolować cały proces powstawania KubePi. Całość jest zintegrowana z portalem Github i przechowywana na prywatnym repozytorium [x].

---

<sup>3</sup>[?]





Rysunek 3.5: Log kontrolny zmian projektowych z portalu Github.

### **Kubernetes Dashboard**

Jest to otwartoźródłowy projekt zapoczątkowany przez firmę Google, który ma na celu umożliwienie zarządzania klastrem Kubernetesa z poziomu przeglądarki. Pozwala użytkownikom na zarządzanie aplikacjami uruchomionymi w klastrze, monitorowanie ich stanu, przeglądanie logów oraz uruchamianie nowych aplikacji.



Rysunek 3.6: Wygląd aplikacji Kubernetes Dashboard.

## 3.3 Projekt

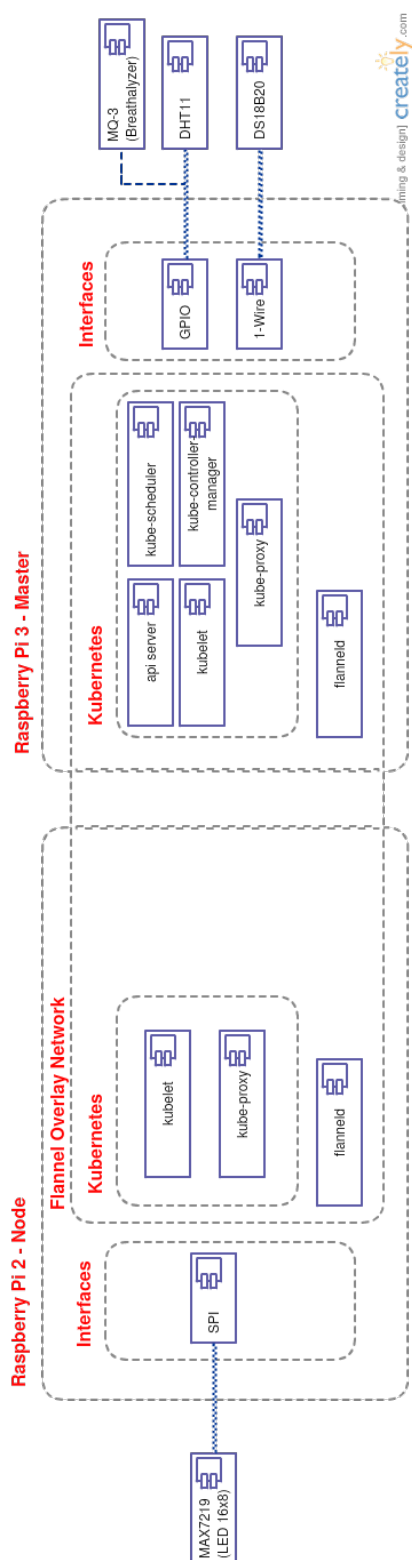
### 3.3.1 Architektura systemu

W tej sekcji przedstawiona zostanie architektura tworzonego systemu oraz schemat połączeń sensorów z płytkami Raspberry Pi. Na poniższym rysunku zobaczyć możemy ogólną architekturę systemu wraz rozkładem uruchomionych procesów.

Na głównym węźle klastra uruchomione są wszystkie procesy zarządzające klastrem czyli:

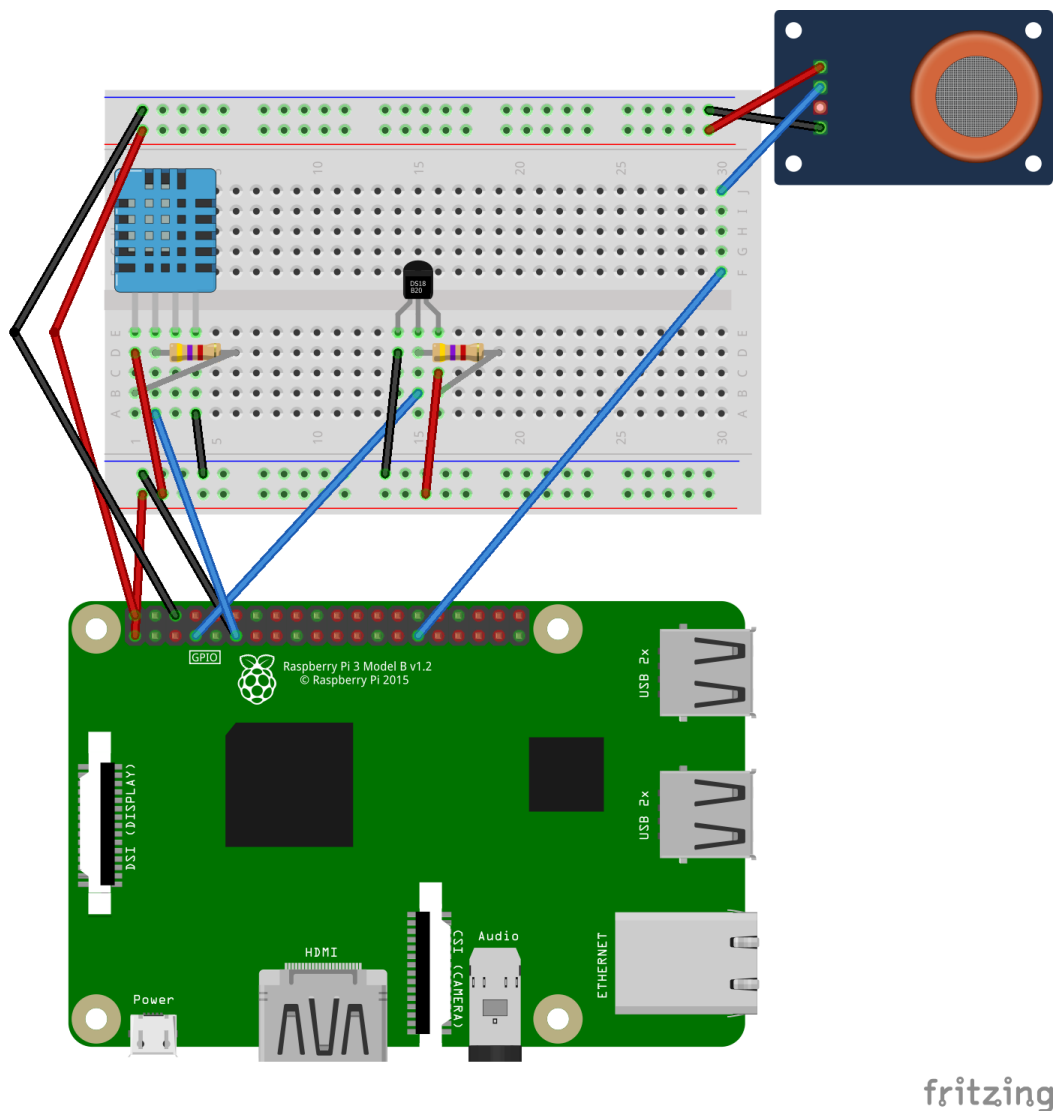
- api-server
- kube-scheduler
- kube-controller-manager
- kube-proxy
- kubelet

Dodatkowo uruchomiony jest proces demon *flanneld*, którego zadaniem jest utworzenie wirtualnej podsieci dla klastra. Możemy również zauważyć podłączone sensory z uwzględnieniem konkretnych interfejsów przy użyciu których komunikują się one z urządzeniem.



Rysunek 3.7: Architektura systemu KubePi.

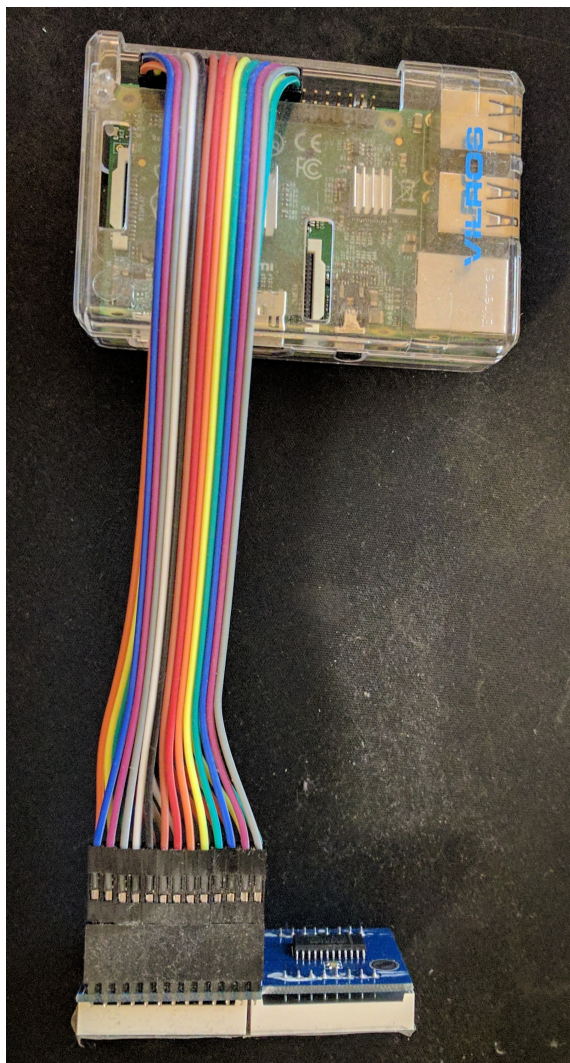
Poniższy schemat obrazuje szczegółowo w jaki sposób konkretne sensory zostały podłączone do mikrokontrolera Raspberry Pi 3. W prawym górnym rogu znajduje się czujnik gazów MQ-3 służący jako wykrywacz alkoholu. Niebieski sensor znajdujący się po lewej stronie to czujnik temperatury i wilgotności DHT11, natomiast najmniejszy sensor znajdujący się na środku to czujnik temperatury DS18B20.



Rysunek 3.8: Schemat podłączenia sensorów do płytki Raspberry Pi 3.

Ostatnie zdjęcie pokazuje podłączenie matrycy LED mającej za zadanie wyświetlać aktualną temperaturę do mikrokontrolera Raspberry Pi 2. Ze względu na

duża ilość kabli zrobione zostało jedynie zdjęcie poglądowe. Użyty moduł można również wpiąć bezpośrednio w płytke Raspberry Pi bez użycia dodatkowych kabli.



Rysunek 3.9: Zdjęcie podłączenia matrycy LED 16x8 MAX7219 do płytki Raspberry Pi 2.

### 3.3.2 Przygotowanie i konfiguracja Raspberry Pi

#### Instalacja systemu

Pierwszym krokiem będzie zainstalowanie systemu HypriotOS. W tym celu należy ściągnąć i wypakować obraz systemu z oficjalnej strony: <https://blog.hypriot.com/downloads/>. Dla potrzeb pracy użyty został system w wersji 1.1.3 jednak nowsze obrazy również powinny działać. W wersji 1.1.3 występuje problem z interfejsem *I-Wire*, który trzeba ręcznie naprawić po instalacji. Możliwe, że w nowszych wersjach problem ten jest również obecny.

Następnym krokiem będzie instalacja obrazu na karcie SD. Bardzo dobrym programem, który dostępny jest na większość systemów jest Etcher dostępny pod adresem: <https://blog.hypriot.com/downloads/>.

Przed pierwszym uruchomieniem możemy zmodyfikować nazwę hosta systemu oraz opcjonalnie ustawić domyślne połączenie z siecią WiFi. W tym celu należy wyszukać na urządzeniu plik o nazwie *device-init.yaml*, a następnie zmienić *hostname* oraz jeśli chcemy aby urządzenie używało sieci WiFi odkomentować i zmienić domyślne ustawienia. Poniżej możemy zobaczyć przykładową zawartość pliku *device-init.yaml*.

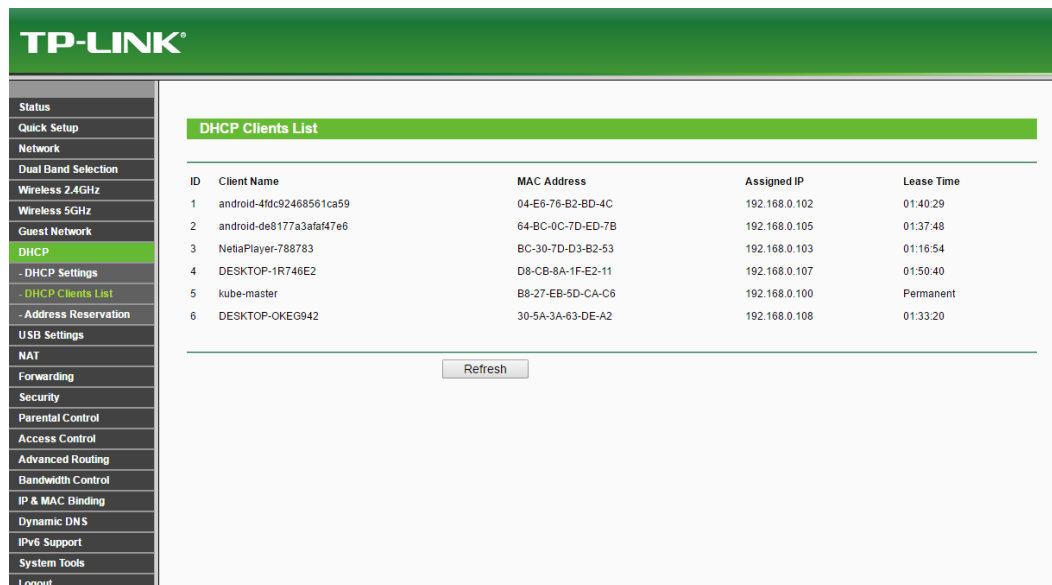
```
# hostname for your HypriotOS device
hostname: kube-master

# optional wireless network settings
wifi:
  interfaces:
    wlan0:
      ssid: "test"
      password: "test"
```

#### Pierwsze uruchomienie

W celu wykonania następnych operacji musimy mieć dostęp do systemu zainstalowanego na urządzeniu. Jedną z możliwości jest podłączenie myszy, klawiatury oraz wyświetlacza HDMI do naszego urządzenia jednak nie jest to zbyt wygodne rozwiązanie. Drugą opcją jest znalezienie adresu IP jaki został przydzielony urządzeniu i komunikacja przy użyciu protokołu SSH. Na systemach linux do znalezienia IP możemy użyć polecenia *nmap*. Więcej informacji o jego użyciu można znaleźć na stronie systemu HypriotOS. Najprostszym sposobem jednak jest zalogowanie się na stronę naszego domowego routera i sprawdzenie listy podłączonych klientów.

Nasze urządzenie powinno identyfikować się ustawioną wcześniej nazwą hosta. W moim przypadku jest to nazwa *kube-master*, a urządzeniu przydzielony został adres IP *192.168.0.100*.



The screenshot shows the TP-Link router's web interface. On the left is a sidebar menu with various settings. The main area displays the 'DHCP Clients List' table, which contains six entries. The fifth entry, 'kube-master', is highlighted in green. Below the table is a 'Refresh' button.

ID	Client Name	MAC Address	Assigned IP	Lease Time
1	android-4f0c92468561ca59	04-E6-76-B2-BD-4C	192.168.0.102	01:40:29
2	android-de8177a3af47e6	64-BC-0C-7D-ED-7B	192.168.0.105	01:37:48
3	NetiaPlayer-788783	BC-30-7D-D3-B2-53	192.168.0.103	01:16:54
4	DESKTOP-1R746E2	D8-CB-8A-1F-E2-11	192.168.0.107	01:50:40
5	kube-master	B8-27-EB-5D-CA-C6	192.168.0.100	Permanent
6	DESKTOP-OKEG942	30-5A-3A-63-DE-A2	192.168.0.108	01:33:20

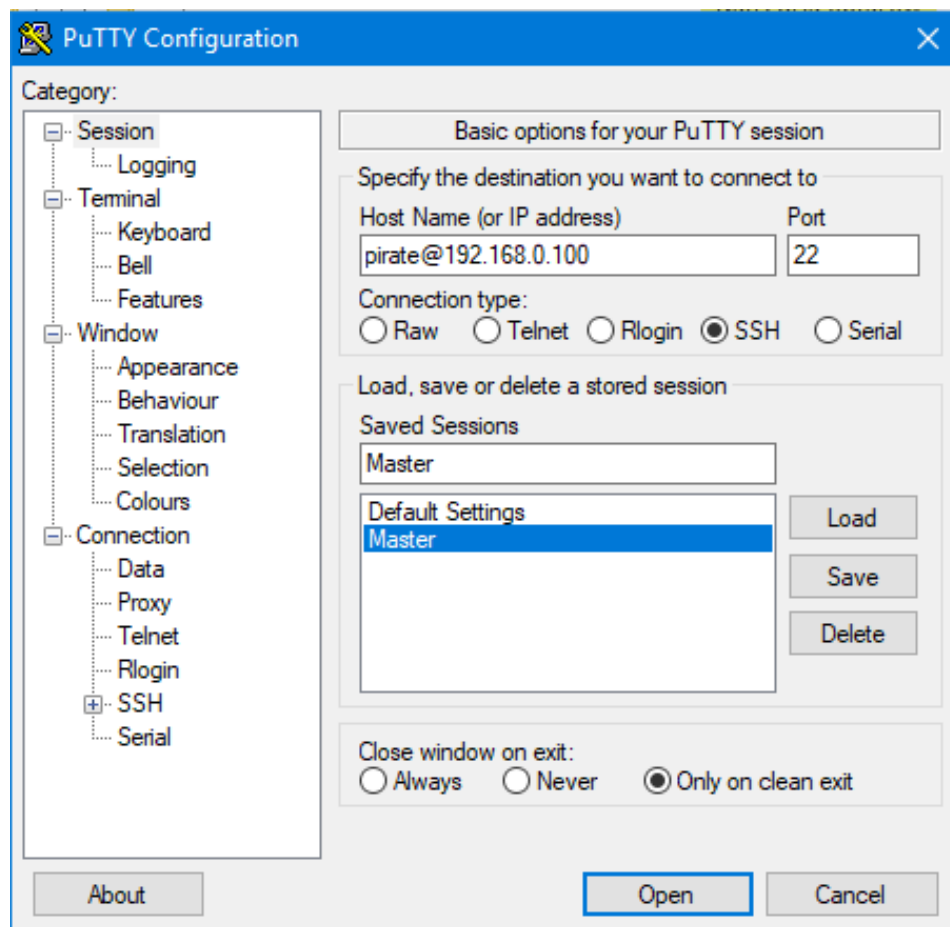
Rysunek 3.10: Konsola admina routera TP-Link.

Do połączenia z urządzeniem wykorzystam darmowy program *putty*<sup>4</sup> służący właśnie do komunikacji z innymi systemami używając między innymi protokołu SSH. Domyślnie dane logowania to *pirate:hyprriot*. Poniżej możemy zobaczyć zrzut ekranu z programu *putty*.

---

<sup>4</sup>putty - strona domowa





Rysunek 3.11: Zrzut ekranu z programu putty.

### Konfiguracja systemu

Najpierw należy uruchomić interfejs *I-wire*. W tym celu należy wyedytować plik */boot/config.txt* i dodać do niego:

```
dtoverlay=w1-gpio
```

Następnie należy uruchomić ponownie urządzenie. Jeśli w folderze */sys/bus* nie pojawi się folder *w1* oznacza to, że najprawdopodobniej w tej wersji systemu również występuje błąd z ładowaniem sterownika interfejsu *I-wire*. Aby go naprawić należy wykonać następującą komendę i ponownie uruchomić system.

```
sudo mv /boot/overlays/w1-gpio.dtbo /boot/overlays/w1-gpio.dtb
```

### 3.3.3 Instalacja i konfiguracja Kubernetesa

W celu zainstalowania Kubernetesa skorzystamy z narzędzia o nazwie *kubeadm*, które zostało stworzone w celu uproszczenia procesu instalacji i konfiguracji klastra. Zaczniemy od dodania repozytoriów oraz instalacji *kubeadm*. Wszystkie komendy należy wykonywać jako *root*.

```
$ apt-get update && apt-get install -y apt-transport-https
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ apt-get update
$ apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

Kolejnym krokiem jest inicjalizacja głównego węzła klastra.

```
$ kubeadm init --pod-network-cidr 10.244.0.0/16
```

Ze względu na pobieranie dość dużej ilości proces ten może potrwać kilkanaście minut. Po zakończeniu uruchamiania należy zapisać sobie polecenie służące do dołączania dodatkowych węzłów do klastra. Poniżej możemy zobaczyć przykładowe polecenie. Token zostaje automatycznie wygenerowany i wygasa po określonym czasie.

```
$ kubeadm join --token 312a65.2cc5af081672daa6
192.168.0.100:6443
```

Następnie ustawiamy narzędzie *kubectl*, aby komunikowało się z naszym klastrem.

```
$ mkdir /home/pirate/.kube
$ sudo cp /etc/kubernetes/admin.conf /home/pirate/.kube/config
$ sudo chown -R pirate /home/pirate/.kube
```

Następnie już jako użytkownik *pirate* możemy sprawdzić stan klastra.

Domyślnie główny węzeł klastra jest zablokowany i niemożliwe jest uruchamianie na nim aplikacji. Należy usunąć tę blokadę następującym poleceniem:

```
$ kubectl get node
NAME                STATUS    AGE           VERSION
kube-master        NotReady  11m           v1.6.1
```

```
$ kubectl taint nodes --all node-role.kubernetes.io/
master-
```

Kolejnym krokiem jest uruchomienie zarządcy sieci, czyli *flannela* w celu umożliwienia komunikacji między aplikacjami w klastrze. Dzięki wykorzystaniu zalet Kubernetesa i stworzeniu *Daemon Seta* na każdym węźle klastra automatycznie zostanie uruchomiona ta aplikacja.

```
$ kubectl create -f https://raw.githubusercontent.com/
coreos/flannel/master/Documentation/kube-flannel-
rbac.yml
$ kubectl create -n kube-system -f https://raw.
githubusercontent.com/floreks/KubePi/master/config/
flannel-ds-arm.yaml
```

Ostatnim krokiem będzie uruchomienie Dashboardu w celu ułatwienia zarządzania klastrem oraz oznaczenie węzła odpowiednią etykietą dzięki czemu możliwe będzie uruchamianie aplikacji na wybranych przez nas węzłach.

Od teraz możemy zarządzać naszym klastrem z poziomu przeglądarki, w naszym przypadku Dashboard dostępny jest pod adresem *192.168.0.100:31143*.

```

$ kubectl label nodes kube-master target=master
node "kube-master" labeled
$ kubectl create -f https://raw.githubusercontent.com/
  floreks/KubePi/master/config/dashboard-arm.yaml

# Czekamy na uruchomienie Dashboardu
$ kubectl -n kube-system get pod
NAME                                READY
  STATUS
etcd-kube-master                    1/1
  Running
kube-apiserver-kube-master          1/1
  Running
kube-controller-manager-kube-master 1/1
  Running
kube-dns-279829092-1fdp6            3/3
  Running
kube-flannel-ds-2ngnl               2/2
  Running
kube-proxy-2kx8h                    1/1
  Running
kube-scheduler-kube-master          1/1
  Running
kubernetes-dashboard-head-2455883860-974t0 1/1
  Running

# Sprawdzamy port na którym wystawiony został Dashboard
$ kubectl -n kube-system get service kubernetes-
  dashboard-head -o template --template="{{ (index .
  spec.ports 0).nodePort }}"
31143

```

The screenshot shows the Kubernetes Dashboard interface. On the left is a sidebar with navigation links: Cluster, Namespaces, Nodes, Persistent Volumes, Roles, Storage Classes, Namespace (All namespaces), Workloads (selected), Daemon Sets, Deployments, Jobs, Pods, Replica Sets, Replication Controllers, Stateful Sets, Discovery and Load Balancing, Ingresses, Services, Config and Storage, Config Maps, Persistent Volume Claims, Secrets, and About. The main content area is titled 'Workloads' and contains three sections: Daemon Sets, Deployments, and Pods.

Daemon Sets						
Name	Namespace	Labels	Pods	Age	Images	
kube-flannel-ds	kube-system	app: flannel tier: node	0 / 1	11 minutes	quay.io/coreos/flannel-v0.7.1-arm	
kube-proxy	kube-system	k8s-app: kube-proxy	0 / 1	13 minutes	gcr.io/google_containers/kube-proxy...	

Deployments						
Name	Namespace	Labels	Pods	Age	Images	
kubernetes-dashboard-head	kube-system	app: kubernetes-dashboard-head	1 / 1	10 minutes	kubernetesdashboard/kubernetes-da...	
kube-dns	kube-system	k8s-app: kube-dns	1 / 1	13 minutes	gcr.io/google_containers/k8s-dns-dnam...	

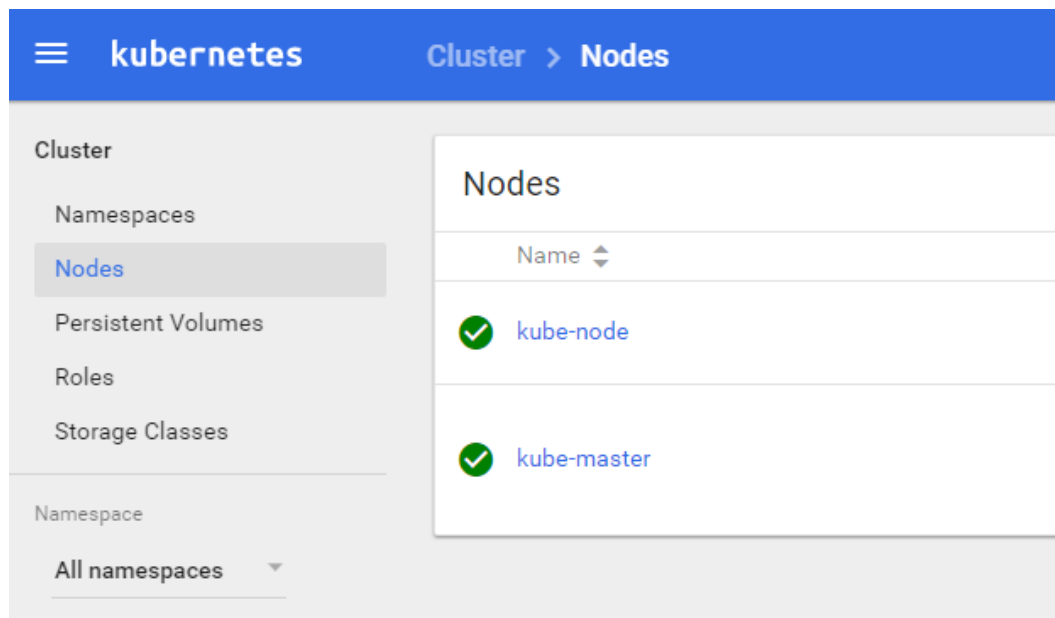
Pods						
Name	Namespace	Status	Restarts	Age		
kubernetes-dashboard-head-2455883860-1chjj	kube-system	Running	0	10 minutes		
kube-flannel-ds-b4ttz	kube-system	Running	0	11 minutes		
kube-apiserver-kube-master	kube-system	Running	6	12 minutes		
kube-controller-manager-kube-master	kube-system	Running	3	12 minutes		
etcd-kube-master	kube-system	Running	0	12 minutes		
kube-scheduler-kube-master	kube-system	Running	2	12 minutes		
kube-dns-279829092-yh44	kube-system	Running	0	13 minutes		
kube-proxy-fpsl9	kube-system	Running	0	13 minutes		

Rysunek 3.12: Zrzut ekranu z aplikacji Kubernetes Dashboard.

Po poprawnym skonfigurowaniu głównego węzła, należy skonfigurować dodatkowy węzeł. W naszym przypadku rolę tę pełnić będzie urządzenie Raspberry Pi 2. Ponownie wypalamy obraz systemu HypriotOS, uruchamiamy i podłączamy się używając protokołu SSH. Należy pamiętać o ustawieniu innej nazwy hosta w pliku *device-init.yaml*. Na potrzeby pracy drugi węzeł nazwany został *kube-node*. Na tej maszynie również należy zainstalować narzędzie *kubeadm* w celu podłączenia węzła do klastra. Kiedy wszystko jest gotowe wykonujemy polecenie:

```
$ kubeadm join --token 645058.48480ab896795f04
192.168.0.100:6443
```

Po chwili nasz węzeł powinien zostać skonfigurowany i dołączony do klastra. Z poziomu Dashboardu możemy sprawdzić jego status.



Rysunek 3.13: Zrzut ekranu pokazujący status węzłów klastra.

Na koniec oznaczamy nowy węzeł odpowiednią etykietą w celu łatwiejszego zarządzania uruchamianiem aplikacji na konkretnych węzłach.

```
$ kubectl label nodes kube-node target=node-1  
node "kube-node" labeled
```

### 3.3.4 Projekt aplikacji przykładowych

W tej sekcji zapoznamy się ze stworzonymi aplikacjami monitorującymi otoczenie. Zwrócimy uwagę na kluczowe elementy implementacji oraz sposób konfiguracji i dystrybucji, który ma na celu uproszczenia procesu uruchamiania aplikacji w klastrze.

#### Monitoring Server

Pierwsza aplikacja pełni rolę serwera do monitorowania temperatury i wilgotności. Pozwala na odczyt pomiarów z opisanych wcześniej sensorów DHT11 oraz DS18B20.

Listing 3.1 przedstawia konfigurację całej aplikacji. Dzięki użyciu biblioteki *restful-go* w prosty sposób można stworzyć serwer bazujący na architekturze REST.

Listing 3.1: Konfiguracja aplikacji

```

var (
    argPort = pflag.Int("port", 3000, "The port to
        listen on for incoming HTTP requests")
)

func main() {
    ...
    // Parse command line arguments
    pflag.CommandLine.AddGoFlagSet(flag.CommandLine)
    pflag.Parse()

    // Register handler
    restful.Add(dht.NewDHT11Service().Handler())
    restful.Add(ds18b20.NewDS18B20Service().Handler())

    log.Printf("Listening on port: %d", *argPort)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%d",
        *argPort), nil))
}

```

Listing 3.2 przedstawia rejestrowanie odpowiednich podstron, które pozwolą na wykonanie kodu poprzez przejście na odpowiedni adres w przeglądarce. W tym wypadku rejestrujemy podstronę `/dht11`, która zwróci nam odczyt z sensora w formacie *JSON*.

Listing 3.2: Konfiguracja API

```

func (d DHT11Service) Handler() *restful.WebService {
    ws := new(restful.WebService)
    ws.
        Path("/dht11").
        Consumes(restful.MIME_JSON).
        Produces(restful.MIME_JSON)

    ws.Route(ws.GET("/").To(d.readFromSensor).
        Doc("Reads temperature and humidity
            from DHT11 sensor").
        Writes(sensor.DHTResponse{}))
}

```

Na listingu 3.3 widzimy kod, który zostaje wykonany po przejściu na podstronę *"/dht11"*. Odczyt danych z sensora zostaje oddelegowany do osobnego interfejsu, a w tej części następuje jedynie obsługa błędów i zwrócenie wyniku.

Listing 3.3: Wywołanie kodu API

```
func (d DHT11Service) readFromSensor(request *restful.
    Request, response *restful.Response) {
    result, err := d.reader.ReadFromSensor()
    if err != nil {
        service.HandleInternalServerError(
            response, err)
        return
    }

    response.WriteHeaderAndEntity(http.StatusOK,
        result)
}
```

Listing 3.4 pokazuje struktury, które zostały zdefiniowane w celu lepszego podziału danych zwracanych przez sensor *DHT11*. *DHTResponse* jest strukturą zwracaną bezpośrednio przez serwer i składa się z temperatury oraz wilgotności. Dodatkowy podział został wprowadzony ze względu na dodanie opcji odczytu pojedynczych wartości temperatury lub wilgotności. Interfejs *DHTReader* natomiast posiada odpowiednie metody do odczytu danych z sensora. Implementacja metod dla sensora *DHT11* znajduje się w osobnym pliku. Dzięki wykorzystaniu interfejsu możliwe jest użycie tych samych struktur do odczytu danych z podobnych sensorów.

Listing 3.4: Struktury odpowiedzi serwera

```
type DHTResponse struct {
    DHTTemperature
    DHTHumidity
}

type DHTTemperature struct {
    // Temperature
    Temperature float32 `json:"temperature"`
}

type DHTHumidity struct {
```



```

        // Humidity
        Humidity float32 `json:"humidity" `
    }

    type DHTReader interface {
        ReadFromSensor() (*DHTResponse, error)
        ReadTemperature() (*DHTTemperature, error)
        ReadHumidity() (*DHTHumidity, error)
        SetGPIO(int)
        ResetGPIO()
    }

```

Ostatni listing 3.5 przedstawia wykorzystanie biblioteki *go-dht* i bezpośredni odczyt wartości sensora *DHT11* z domyślnie ustawionego 17 pinu GPIO.

Listing 3.5: Odczyt danych z sensora DHT11

```

func (DHT11Reader) ReadFromSensor() (*DHTResponse,
    error) {
    log.Printf("Reading from sensor %s on GPIO pin %d", dht.DHT11, gpioPinOverride)
    temp, humidity, _, err := dht.
        ReadDHTxxWithRetry(dht.DHT11,
            gpioPinOverride, BOOST_PERF_FLAG,
            RETRIES)
    if err != nil {
        if strings.Contains(err.Error(), "C.
            dial_DHTxx_and_read") {
            return nil, fmt.Errorf("Could not read from sensor.")
        }
    }

    return nil, err
}

return &DHTResponse{DHTTemperature{Temperature:
    temp}, DHTHumidity{Humidity: humidity}},
    nil
}

```

Aplikacja posiada również opcję nadpisania numeru pinu GPIO, z którego należy

odczytać dane. Możliwe jest to dzięki wywołaniu operacji *POST* na podstronie o adresie */dht11* i przesłanie parametru *gpio=x*, gdzie *x* to numer pinu GPIO z którego chcemy odczytywać dane.

### Breathalyzer

Kolejna aplikacja ma za zadanie wykrywanie alkoholu w powietrzu. Po uruchomieniu mierzy przez 5 sekund zawartość alkoholu, następnie uśrednia wynik i informuje użytkownika czy alkohol został wykryty czy nie. Nie posiada ona funkcji dokładnego pomiaru stężenia alkoholu, a jedynie informuje o jego obecności.

Listing 3.6 podobnie do listingu 3.2 przedstawia budowanie API naszej aplikacji. Przejście na postronę */mq3/measure* uruchamia pomiar.

Listing 3.6: Konfiguracja API

```
func (d MQ3Service) Handler() *restful.WebService {  
    ws := new(restful.WebService)  
    ws.  
        Path("/mq3").  
        Consumes(restful.MIME_JSON).  
        Produces(restful.MIME_JSON)  
  
    ws.Route(ws.GET("/measure").To(d.detectAlcohol)  
        .  
        Doc("Reads temperature from DS18B20 sensor").  
        Writes(sensor.MQ3Response{}))  
  
    return ws  
}
```

Listing 3.7 pokazuje główną metodę całej aplikacji, której zadaniem jest sprawdzenie obecności alkoholu w powietrzu. Pomiar odbywa się w 5-sekundowym przedziale czasowym z częstotliwością 10 odczytów na sekundę, co pozwala na uzyskanie 50 próbek danych. Dane te są następnie uśredniane, a wykrycie alkoholu sygnalizowane w przypadku jeśli czujnik pozostawał w stanie niskim co najmniej przez połowę okresu próbkowania.

Listing 3.7: Pomiar alkoholu w powietrzu

```

// DetectAlcohol reads for 5 sec digital out from mq-3
// sensor and returns true or false based on readings
// over time
func (this MQ3Reader) DetectAlcohol() MQ3Response {
    timeoutChan := time.NewTimer(this.
        detectionTimeout).C
    intervalChan := time.NewTicker(this.
        detectionInterval).C
    var result int
    tmp := 1.0
    readings := this.detectionTimeout.Seconds() /
        this.detectionInterval.Seconds()
    pin := rpi.Pin(this.gpioPin)
    // Set input mode to read from the pin
    pin.Input()

    for {
        select {
            case <-timeoutChan:
                tmp = tmp / float64(readings)
                result = int(tmp + 0.5)

                if result == 0 {
                    return MQ3Response{
                        AlcoholDetected:
                            false}
                }

                return MQ3Response{
                    AlcoholDetected: true}
            case <-intervalChan:
                out := pin.Read()
                if out == rpi.Low {
                    tmp += 1.0
                }
        }
    }
}

```

### Wyświetlacz temperatury

Zadaniem tej aplikacji jest okresowe pobieranie temperatury poprzez odpytywanie stworzonego wcześniej serwera monitorującego oraz wyświetlanie jej na matrycy LED MAX7219.

Listing 3.8 przedstawia konfigurację aplikacji. Aplikacja przyjmuje jako parametr adres serwera, z którego pobiera potrzebne dane. W dalszym kroku następuje konfiguracja wyświetlacza oraz startowany jest wątek odpytujący serwer. Komunikacja między wątkami odbywa się poprzez kanały, które są podstawowymi mechanizmami komunikacji między wątkami w języku Go. Po odczytaniu danych z serwera są one zapisywane do kanału *tempChan*, a następnie główny wątek pobiera te dane i wyświetla temperaturę.

Listing 3.8: Konfiguracja aplikacji

```
func main() {  
    dev, err := device.NewMax7219()  
    if err != nil {  
        ...  
    }  
    err = dev.Init()  
    if err != nil {  
        ...  
    }  
  
    defer dev.Close()  
    tempChan := make(chan float32)  
  
    go pollTemperature(pollServer, tempChan)  
    registerSigtermHandler(dev)  
  
    for temp := range tempChan {  
        err := dev.DisplayTemperature(temp)  
        if err != nil {  
            ...  
        }  
    }  
}
```

Następny listing 3.9 przedstawia proces pobierania temperatury z serwera i zapisywania do struktury *TempResponse*. Co pewien okres czasu, zdefiniowany w zmiennej *POLLING\_TIME* dane pobierane są z serwera. Następnie są one dekodowane do struktury *TempResponse*, która zawiera tylko jedno pole *Temperature*. Po poprawnym zdekodowaniu struktury jest ona przesyłana do kanału *tempChan*, który służy do synchronizacji z głównym wątkiem aplikacji.

Listing 3.9: Odczyt temperatury z serwera

```
func pollTemperature(pollServer *string, tempChan chan
float32) {
    tempResponse := new(TempResponse)

    for {
        r, err := http.Get(*pollServer)
        if err != nil {
            log.Printf("Error_during_
temperature_polling:_%s",
err.Error())
            os.Exit(1)
        }

        err = json.NewDecoder(r.Body).Decode(
tempResponse)
        if err != nil {
            log.Printf("Error_during_server
_response_decoding:_%s", err
.Error())
            os.Exit(1)
        }

        tempChan <- tempResponse.Temperature
        time.Sleep(POLLING_TIME)
        r.Body.Close()
    }
}
```

Ostatnim i najważniejszym krokiem jest wyświetlenie temperatury, co przedstawia kod w listingu 3.10. Temperatura zostaje rzutowana na liczbę całkowitą, ze względu na brak miejsca na wyświetlaczu. Następnie liczba ta konwertowana jest na odpowiednią tablicę bajtów o rozmiarze 2x8. Każda wiersza reprezentuje cyfrę, która wyświetlana jest na obszarze 8x8. Każda kolumna reprezentuje natomiast pojedynczy wiersz 8 diód na matrycy. Przykładowo wpisanie cyfry 2 (binarnie 00000010) do rejestru 0 spowoduje podświetlenie przedostatniej diody w pierwszym wierszu matrycy LED.

Listing 3.10: Wyświetlanie temperatury

```
func (this Max7219) DisplayTemperature(temperature
    float32) error {
    // Round to integer number as we don't have
    // space for floating point numbers
    temp := int(temperature)

    buf, err := DEFAULT_FONT.FromInt(temp)
    if err != nil {
        return err
    }

    for i := 0; i < MAX7219_DIGIT_COUNT; i++ {
        err := this.send(Max7219Reg(i), buf[1][
            i], buf[0][i])
        if err != nil {
            return err
        }
    }

    return nil
}
```

### KubePi Monitor

Ostatnia aplikacja jest aplikacją webową i pozwala na monitorowanie odczytów sensorów z poziomu przeglądarki. Podobnie do poprzedniej aplikacji pobiera okresowo dane z serwera monitorującego i wyświetla je oraz na żądanie użytkownika startuje wykrywacz alkoholu.

Listing 3.11 przedstawia funkcję służącą do startowania wątku, który okresowo (domyślnie co 3 sekundy) odpytuje serwer przekazany jako parametr funkcji o dane.

Listing 3.11: Startowanie wątków do odczytu danych

```
runPollInterval(fetchFn, address, timeout = 3000,
  immediate = true) {
  let intervalId = setInterval(() => {
    this.props.dispatch(fetchFn(address))
  }, timeout, true)

  if(immediate) {
    this.props.dispatch(fetchFn(address))
  }

  return intervalId
}
```

Funkcja przedstawiona na listingu 3.12 jest odpowiedzialna za uruchamianie oraz zatrzymywanie wątku pobierającego dane z czujnika temperatury i wilgotności DHT11. Odpytuje ona lokalny serwer o dane, który jest pośrednikiem między częścią webową, a klastrem.

Listing 3.12: Startowanie wątku odpytującego nasz serwer o dane z sensora DHT11

```
handleDHT11() {
  if(this.state.dhtIntervalId === 0) {
    let intervalId = this.runPollInterval(dht11.
      fetchData, '/dht11')
    this.setState({
      dhtIntervalId: intervalId,
    })
  } else {
    clearInterval(this.state.dhtIntervalId)
  }
}
```

```

        this.props.dispatch(dht11.reset())
        this.setState({
            dhtIntervalId: 0,
        })
    }
}

```

Ostatni listing 3.13 przedstawia funkcję serwera Go, która odpowiedzialna jest za pobieranie odczytów z serwera monitorującego opisanego w sekcji 3.3.4.

Listing 3.13: Odczyt temperatury z naszego serwera

```

func dht11Handler(w http.ResponseWriter, r *http.
    Request) {
    dht11Response := new(DHT11Response)

    resp, err := http.Get("http://dht-server.
        default.svc.cluster.local:3000/dht11")
    if err != nil {
        fmt.Fprintf(w, "Error_during_
            temperature_polling:_%s", err.Error
                ())
        return
    }

    err = json.NewDecoder(resp.Body).Decode(
        dht11Response)
    if err != nil {
        fmt.Fprintf(w, "Error_during_server_
            response_decoding:_%s", err.Error())
        return
    }

    log.Println(dht11Response)
    w.Header().Set("Content-Type", "application/
        json")
    json.NewEncoder(w).Encode(dht11Response)
}

```



## Budowanie i dystrybucja

W tej sekcji opisane zostały proces budowania i dystrybucji aplikacji wraz z użytymi narzędziami.

Język Go pozwala na łatwe budowanie aplikacji do pojedynczego pliku wykonywalnego. Użycie aplikacji *xgo*, pozwoliło uprościć proces budowania aplikacji na różne architektury. Budowanie aplikacji na inne architektury odbywa się w kontenerze Dockera, który został wcześniej skonfigurowany i zawiera wszystkie zależności potrzebne do budowania aplikacji. Domyślnie korzysta on z kompilatora języka Go, jednak używa zaawansowanej konfiguracji w celu zoptymalizowania procesu kompilacji.

Listing 3.14 przedstawia plik *Makefile* użyty do budowania aplikacji opisanej w sekcji 3.3.4 zgodnej z architekturą ARM. Pozostałe aplikacje korzystają z podobnej konfiguracji.

Listing 3.14: Makefile użyty do budowania aplikacji dla architektury ARM

```
GO := $(shell command -v go 2> /dev/null)
XGO := $(shell command -v xgo 2> /dev/null)

prepare:
    @mkdir build

all: prepare arm

arm: prepare
ifndef XGO
    $(error "Could not find XGO compiler.")
endif
    @xgo -targets linux/arm-6 .
    @mv dht* build/

clean:
    @sudo rm -rf build
```

Poniższy listing 3.15 pokazuje prosty plik Dockera, który został użyty do zbudowania kontenera aplikacji webowej opisanej w sekcji 3.3.4. Dodaje on zbudowany plik wykonywalny do kontenera, następnie ustawia go jako domyślnie uruchamiany plik po wystartowaniu kontenera i udostępnia aplikację na porcie 80.

Listing 3.15: Przykładowy plik Dockera dla aplikacji webowej

```
FROM floreks/kubepi-base

ADD build /

ENTRYPOINT ["/kubepi-monitor-linux-arm-6"]

EXPOSE 80
```

Ostatnim krokiem po zbudowaniu lokalnego kontenera jest wysłanie go na darmowy serwer przechowujący aplikacje Dockera, a mianowicie DockerHub. Komendy uruchamiane są z folderu zawierającego plik *Dockerfile*.

```
# Zbudowanie kontenera aplikacji o nazwie dht-server
# dla konta floreks
$ docker build -t floreks/dht-server .
# Wysłanie kontenera aplikacji na serwery DockerHub
$ docker push floreks/dht-server
```

## Uruchamianie w klastrze

W celu uruchomienia naszej aplikacji w klastrze Kubernetesa wymagane jest stworzenie odpowiedniego pliku w formacie *json* lub *yaml*, którego struktura została przedstawiona poniżej. Dzieli się on na kilka części. Pierwsza część widoczna na listingu 3.16 przedstawia ogólną definicję aplikacji. Najważniejszymi elementami są:

- **kind: Deployment** — definiuje typ obiektu, który odpowiedzialny będzie za zarządzanie naszą aplikacją.
- **labels** — definiuje etykiety dla naszej aplikacji. Służą one głównie do powiązania obiektów typu *Pod* z *Deploymentem*.
- **replicas** — definiuje liczbę replik naszej aplikacji, czyli ile kopii naszej aplikacji będzie uruchomionych w klastrze.
- **nodeSelector** — definiuje etykietę węzła na którym może zostać uruchomiona nasza aplikacja. Jest to informacja dla planisty o tym, że dana aplikacja może być uruchomiona tylko na węzłach, które posiadają taką etykietę.

Listing 3.16: Definicja aplikacji uruchamianej w klastrze - część 1

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  labels:
    app: dht-server
  name: dht-server
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dht-server
  template:
    metadata:
      labels:
        app: dht-server
    spec:
      nodeSelector:
        target: master
```

Listing 3.17 pokazuje definicję kontenera aplikacji. Najważniejszymi elementami tej części są:

- **privileged** — pozwala na dostęp aplikacji do plików hosta oraz nadaje jej prawa *roota*.
- **image** — definiuje obraz aplikacji Dockera. Domyślnie pobierany jest on z DockerHuba.
- **volumeMounts** — definiuje ścieżki pod którymi zamontowane zostają systemy plików zdefiniowane w następnej części.
- **volumes** — definiuje systemy plików (foldery, serwery, itp.), które mogą być zamontowane w kontenerze aplikacji.

Listing 3.17: Definicja aplikacji uruchamianej w klastrze - część 2

```
containers:
- name: dht-server
  securityContext:
    privileged: true
  image: florekx/dht-server
  imagePullPolicy: Always
  ports:
  - containerPort: 3000
    protocol: TCP
  volumeMounts:
  - mountPath: /sys/class/gpio
    name: sys-class-gpio
  - mountPath: /sys/bus
    name: sys-bus
volumes:
- name: sys-class-gpio
  hostPath:
    path: /sys/class/gpio
- name: sys-bus
  hostPath:
    path: /sys/bus
```

Listing 3.18 definiuje serwis dla naszej aplikacji, który pozwoli udostępnić ją użytkownikom. Kluczowymi elementami tej części są:

- **type** — definiuje typ naszego serwisu. *NodePort* oznacza, że aplikacja udostępniona będzie na automatycznie przydzielonym porcie z zakresu 30000-32767.
- **ports** — definiuje port lub porty, które mają zostać przekierowane z wnętrza kontenera na zewnątrz. Przykładowo jeśli nasz serwer aplikacji uruchamiany jest na porcie 3000 w kontenerze, to dzięki temu będzie on dostępny również na porcie 3000 na przydzielonym dla Poda adresie IP.

Listing 3.18: Definicja aplikacji uruchamianej w klastrze - część 3

```
kind: Service
apiVersion: v1
metadata:
  labels:
    app: dht-server
  name: dht-server
  namespace: default
spec:
  type: NodePort
  ports:
    - port: 3000
      targetPort: 3000
  selector:
    app: dht-server
```

## 3.4 Opis użycia

## 3.5 Możliwości rozszerzania aplikacji

## **Rozdział 4**

### **Podsumowanie**

# Bibliografia

[1] TODO

# Spis rysunków

2.1	Przedstawienie Internetu Rzeczy w ujęciu graficznym . . . . .	10
2.2	Architektura chmur obliczeniowych . . . . .	13
2.3	Architektura chmur obliczeniowych . . . . .	14
2.4	Architektura wirtualizacji opartej o kontenery . . . . .	16
2.5	Architektura wirtualizacji opartej o kontenery . . . . .	18
2.6	Architektura wirtualizacji opartej o kontenery . . . . .	20
2.7	Architektura systemu Borg . . . . .	23
2.8	Architektura Kubernetesa . . . . .	25
2.9	Architektura Docker Swarm . . . . .	29
2.10	Architektura Resina . . . . .	30
2.11	Mikrokontroler Raspberry Pi . . . . .	31
2.12	Układ pinów mikrokontrolera Raspberry Pi 3 . . . . .	33
3.1	Czujnik temperatury i wilgotności DHT11. . . . .	37
3.2	Czujnik temperatury DS18B20. . . . .	38
3.3	Wyświetlacz LED MAX7219 (16x8). . . . .	38
3.4	Logo języka Go i biblioteki ReactJS . . . . .	39
3.5	Log kontrolny zmian projektowych z portalu Github. . . . .	41
3.6	Wygląd aplikacji Kubernetes Dashboard. . . . .	42
3.7	Architektura systemu KubePi. . . . .	44
3.8	Schemat podłączenia sensorów do płytki Raspberry Pi 3. . . . .	45
3.9	Zdjęcie podłączenia matrycy LED 16x8 MAX7219 do płytki Raspberry Pi 2. . . . .	46
3.10	Konsola admina routera TP-Link. . . . .	48
3.11	Zrzut ekranu z programu putty. . . . .	49
3.12	Zrzut ekranu z aplikacji Kubernetes Dashboard. . . . .	53
3.13	Zrzut ekranu pokazujący status węzłów klastra. . . . .	54



## **Spis tablic**