Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Katedra Informatyki

PRACA MAGISTERSKA

PIOTR KOZŁOWSKI

PRZENOŚNY KLASTER OBLICZENIOWY OPARTY NA URZĄDZENIACH SOC

PROMOTOR:
dr hab. inż. Aleksander Byrski

Kraków 2015

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZĄ PRACĘ DYPLOMOWĄ WYKONAŁEM OSOBIŚCIE I SAMODZIELNIE, I NIE KORZYSTAŁEM ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W PRACY.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Department of Computer Science

MASTER OF SCIENCE THESIS

PIOTR KOZŁOWSKI

MOBILE COMPUTING CLUSTER BASED ON SOC DEVICES

SUPERVISOR:
Aleksander Byrski Ph.D

Krakow 2015

Podziękowania...

Spis treści

1. Wprowadzenie

1.1. Wstęp

Tematyka niniejszej pracy jest związana z przetwarzaniem równoległym w środowisku rozproszonym oraz algorytmami ewolucyjnymi. Obliczenia równoległe, systemy wysokiej dostępności oraz wielkiej skali są tematem bardzo aktualnym i wciąż powszechnie rozwijanym. Powstaje wiele urządzeń oraz systemów wieloprocesorowych zbudowanych w celu połączenia zasobów w jedną całość oraz wykorzystujących w maksymalnym stopniu ich moc obliczeniową. Zasoby te mogą być zlokalizowane lokalnie lub być rozproszone w różnych rejonach świata, czyli np. w sieci Internet. Przykładem mogą być takie projekty jak Hadoop lub Condor. Połączone zasoby mogą zostać wykorzystane do obliczeń o dużej skali (np. symulacji) i przyśpieszenia krytycznych obliczeń (systemy obronne, loty kosmiczne, prognozy). Jak również są przydatne w przypadku przetwarzania dużej liczby rozproszonych danych i rozwiązywania trudnych problemów optymalizacji (przeszukiwania dużych przestrzeni rozwiązań np. w celu poszukiwania ekstremum). Do głównych obszarów zastosowań można zaliczyć też obliczenia naukowe prowadzone przez różne uniwersytety i inne ośrodki badawcze, które w dużym stopniu przyczyniają się do rośnięcia zapotrzebowania na takie rozwiązania oraz powstawanie nowych. We wcześniejszych latach ta tematyka napotykała różne bariery w rozwoju, takie jak zbyt mały rynek zbytu, spowodowany wysokimi kosztami utrzymania oraz realizacji architektury. Było również mniej rozwiązań tego typu oraz mniejsze zapotrzebowanie na nie. Klastry są bardziej elastycznym rozwiązaniem a co za tym idzie bardziej opłacalnym. Stały się zatem nową perspektywą, powszechnie już wykorzystywaną w dzisiejszych systemach. Możliwość użycia tego typu rozwiązań przez zwykłych użytkowników i mniejsze firmy, komercjalizacja, wykorzystanie w przemyśle i elastyczne formy handlu, np. sprzedaż zużytej mocy obliczeniowej w bardzo dużym stopniu przyczyniły się do popularyzacji tejże tematyki. Trwa ciągły rozwój technologii z tym zwiazanych, mając na uwadze takie rzeczy jak wydajność i zapotrzebowanie na energię oraz łatwa dostępność. Wiele problemów nie zostsało jeszcze rozwiązanych lub w wystarczającym stopniu zbadanych. Argumenty stojące za rozwojem tych technologii to m.in. rosnące zapotrzebowanie na moc obliczeniową, stosunek ceny do mocy obliczeniowej, bariery technologiczne rozwoju procesorów sekwencyjnych, dzielenie zasobów, skalowalność, komunikacja.

1.2. Aktualny stan wiedzy

Do tej pory zostało opracowanych kilka klastrów obliczeniowych z wykorzystaniem urządzeń SoC takich jak Raspberry Pi. Przykładem może być praca opisana w artykule Iridis-pi: a low-cost, compact demonstration cluster [41]. Powstało również kilka podobnych rozwiązań o których informacje można znaleść na Internecie. Dosyć dawno ukazało się wydanie czasopisma Informatica opisujące klaster obliczeniowy wykorzystujący telefony komórkowe, oraz rolę komunikacji i kilka różnych podeść bazujących na różnych architekturach systemu [35]. Dwa kolejne artykuły opisują klastry w których skład wchodzą urządzenia przenośne podłączone do sieci Internet lub połączone w szybkiej sieci lokalnej ukazały się na łamach ... [38, 50]. Poruszono tam także problem dynamicznego load-balancingu oraz zrównoleglania obliczeń. W artykule Adaptive clustering for mobile wireless networks [47] opisano samoorganizującą się, odporną na uszkodzenia składowych węzłów mobilną sieć oraz algorytmy z nią

związane, które są dosyć istotne w kontekście niniejszej pracy. W pracy Cluster Computing: A Mobile Code Approach [52] opisano klaster wysokiej wydajności wraz z jego architekturą oraz implementacją. Natomiast praca pt. DroidCluster: Towards Smartphone Cluster Computing - The Streets are Paved with Potential Computer Clusters [56] opisuje nowoczesne smarfony wchodzące w skład klastra obliczeniowego oraz temat Grid-Computing. Jak widać temat i problemy z nim związane znany jest od dosyć dawna. Natomiast wciąż jest aktualny i nadal brakuje skutecznych rozwiązań wszystkich problemów. Same urządzenia SoC zyskały swoją popularność dopiero w ostatnich latach, głównie za sprawą taniego mini komputera jakim jest Raspberry Pi. Większość powstałych rozwiązań bazuje na takich technologiach jak MPI oraz Hadoop, natomiast w niniejszej pracy autor skupi się na technologiach pracujących w wirtualnej maszynie Javy, co jeszcze bardziej zwiększy mobilność powstałej platformy.

1.3. Cel pracy

Do celów niniejszej pracy należy nakreślenie możliwości zastosowania przenośnych klastrów obliczeniowych, przedstawienie problemów związanych z ich wykorzystaniem, zidentyfikowanie obszarów potencjalnych usprawnień oraz wykorzystanie przygotowanego klastra w celu rozwiązania trudnego problemu obliczeniowego. Jako część programistyczna niniejszej pracy, zostanie wykonana mobilna platforma łącząca możliwości urządzeń SoC, nadzorująca ich pracę oraz zarządzająca dostępną energią. Pod uwagę zostanie również wzięta odporność na uszkodzenia oraz łatwa zastępowalność urządzeń. Zostaną wykorzystane takie paradygmaty jak programowanie funkcjonalne oraz reaktywne, z wykorzystaniem technologii Scala oraz Akka wraz z podejściem równoległym. Jako część persystencji danych zostanie wykorzystana baza NoSQL w technologii MongoDB. Część kliencka zostanie zrealizowana w technologii REST/HTTP z użyciem frameworka Play oraz AngularJS. Zaprojektowana platforma będzie w stanie monitorować pracę klastra, "łączyć" komponenty w jedną całość czyli zarządzać komunikacją, być odporna na sytuację uszkodzenia jednego z elementów klastra oraz częściową utratę danych, maksymalnie wykorzystywać moc obliczeniową elementów klastra oraz zarządzać load balancingiem. Niniejsza praca będzie próbą wykorzystania urządzeń SoC do wykonywania wydajnych obliczeń prowadzonych w wirtualnej maszynie Javy oraz próbą odpowiedzenia na pytanie czy jest możliwe zaprogramowanie komunikacji między elementami klastra w sieci Ethernet nie powodującej zbyt wielkiego narzutu przesyłu danych w stosunku do szybkości obliczeń. Praca będzie też próbą wykorzystania wirtualnej maszyny Javy oraz technologii wysokiego poziomu do wykonywania obliczeń dużej skali i sprawdzenie wydajności takich rozwiązań. Stworzenie samoorganizującej się oraz inteligentnej platformy będzie bardzo dużym wyzwaniem wkraczającym trochę w tematykę AI oraz Systemów Agentowych. Jeśli rozwiązanie wykorzystujące technologie przedstawione w niniejszej pracy będzie wystarczająco wydajne i odpowiednio skalowalne, ułatwi to w znacznym stopniu mobilność tego rozwiązania oraz rozwiąże potrzebę tworzenia nowych bibliotek z myślą o konkretnej platformie. Co za tym idzie większą część czasu i nakładów finansowych będzie można poświęcać na implementację rozwiązania samego problemu aniżeli na uruchomienie tej implementacji lub tworzenie rozwiązań pasujących do konkretnego sprzętu (architektury) lub systemu.

2. Charakterystyka rozpatrywanego problemu

W poniższym rozdziale zarysowano kontekst problemów poruszanych w niniejszej pracy.

2.1. Urządzenia SoC

2.2. Cluster Computing

2.3. Obliczenia równoległe oraz rozproszone

2.4. Istniejące rozwiązania

3. Opis rozwiązania

Niniejszy rozdział zawiera charakterystykę rozwiązania stworzonego na potrzeby niniejszej pracy, przedstawiono tutaj najważniejsze wykorzystane algorytmy oraz omówiono w skrócie użyte technologie.

3.1. Wykorzystane algorytmy

W rozdziale opisano kilka algorytmów użytych podczas implementacji rozwiązania stworzonego na potrzeby niniejszej pracy.

3.1.1. Algorytmy genetyczne

Algorytmy genetyczne zaliczają się do grupy algorytmów ewolucyjnych. Zostały opracowane przez Johna Hollanda w latach siedemdziesiątych i są inspirowane teorią ewolucji Darwina tudzież ewolucją biologiczną [42]. Istnieje wiele różnych prac traktujących o wykorzystaniu algorytmów genetycznych, różnych ich odmianach czy modyfikacjach oraz badaniu ich efektywności. Przykładem może być praca magisterska zawarta w pozycji [42] lub trochę starsza pozycja książkowa [44]. Opis implementacji algorytmów genetycznych można znaleść w pozycji [54].

Pojęcia

- Gen - cecha mająca wpływ na jakość rozwiązania.
- Chromosom - indywiduum, osobnik prezentujący jedno z rozwiązań problemu, składa się z genów.
- Populacja - zbiór chromosomów, prezentuje zbiór rozwiązań danego problemu.
- Migracja - wymiana osobników pomiędzy populacjami.
- Selekcja - algorytm wyboru osobników pozostałych w danej populacji w następnym cyklu.
- Krzyżowanie - tworzenie osobników potomnych w trakcie trwania ewolucji na podstawie innych, wybranych osobników - rodziców.
- Mutacja - modyfikacja osobnika zawierająca element losowości, mająca na celu próbę ekploracji nowych obszarów rozwiązań.
- Funkcja przystosowania (ang. fitness) - zwana też funckją celu, określa jakość danego rozwiązania.
- Ewaluacja - wyliczenie wartości funkcji fitness dla każdego osobnika populacji czyli inaczej ocena jakości znalezionych rozwiązań.
- Ewolucja - powtarzający się cykl w trakcie którego następuje rozwój populacji oraz w skład którego wchodzi m.in. ewaluacja, selekcja, krzyżowanie, mutacja oraz migracja.
- Generacja - jeden cykl algorytmu.

Model wyspowy

Jest to odmiana algorytmu genetycznego przystosowana do obliczeń równoległych oraz rozproszonych. Zamiast jednej dużej populacji rozważamy tutaj kilka podpopulacji zwanych wyspami. Każda wyspa może być traktowana jako osobna populacja, ponieważ ewolucja na niej zachodzi w izolacji od pozostałych wysp. Co pewien okres populacje znajdujące się na wyspach wymieniają się między sobą pewną ilością osobników w procesie migracji. Pozwala to przyśpieszyć znajdywanie rozwiązania. Istotna jest tutaj topologia połączeń pomiędzy wyspami pozwalająca na wymianę osobników, metoda selekcji migrujących osobników, ich wielkość oraz odstęp pomiędzy kolejnymi migracjami. Badania różnych parametrów tych zjawis można znaleść m.in. w pracy [42].

Zastosowanie

Algorytmy ewolucyjne sprawdzają się tam gdzie nie jest dobrze znana przestrzeń rozwiązań ale został określony sposób oceny jakości rozwiązania. Znajdują one zastosowanie przy rozwiązywaniu problemów NP, np. problemu komiwojażera w którym trzeba znaleść najkrótszą drogę łączącą wszystkie miasta tak, aby odwiedzić każde miasto conajmniej raz. Innymi zastosowaniami mogą być choćby poszukiwania ekstremów funkcji, których nie da się obliczyć analitycznie lub przeszukiwanie dużych przestrzeni rozwiązań jak np. w przypadku problemu grupowania. Alborytmy genetyczne są mniej zależne od wstępnej inicjalizacji zadania oraz mniej skłonne do znajdywania rozwiązań lokalnych zamiast optymalnych [9, 42].

3.1.2. Master/slave

Master/slave jest modelem komunikacji w którym jedno urządzenie lub proces zwane master kontroluje jedno lub więcej urządzeń lub procesów zwanych slave lub worker [22].

Ten model komunikacji może zostać wykorzystany w następujących przypadkach:

- gdy chcemy mieć jeden proces, tzw. single-point of responsibility, który podejmuje decyzje lub koordynuje akcje w celu zachowania spójności w systemie,
- potrzebujemy zapewnić jeden punkt dostępu do zewnętrznego systemu lub jeden punkt wejścia do systemu,
- istnieje potrzeba stworzenia scentralizowanego serwisu, np. zajmującego się routingiem.

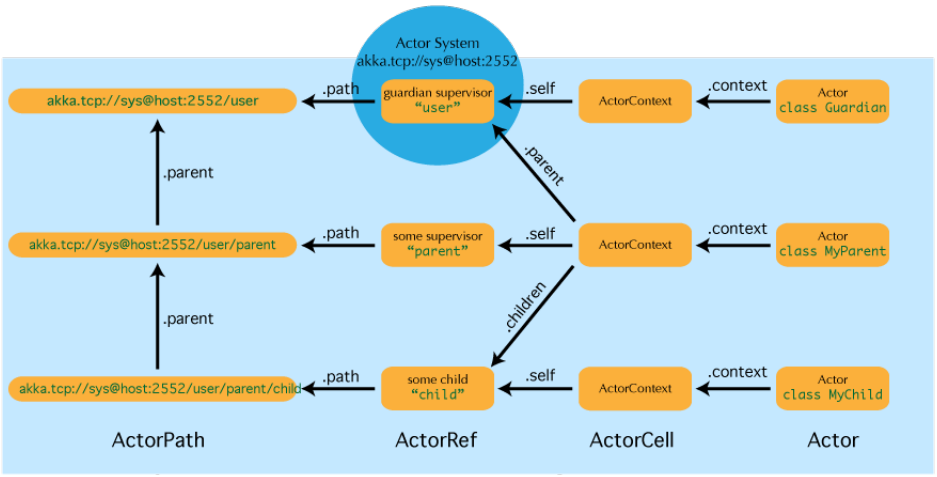
Rozwiązanie to ma niestety również kilka wad, jak na przykład istnieje niebezpieczeństwo utworzenia tzw. wąskiego gardła co może spowodować problemy z wydajnością lub być tzw. single-point of failure. Jeśli urządzenie/proces posiadające rolę master ulega awarii, powinniśmy w jakiś sposób obsłużyć taką sytuację, np. poprzez ponowne wystartowanie procesu/urządzenia z rolą master aby zapewnić poprawne i niezawodne działanie komunikacji. Sytuacje takie mogą wprowadzić pewne opóźnienia w trakcie odzyskiwania sprawności systemu [17].

3.2. Użyte technologie

Podrozdział przedstawia krótką charakterystykę wykorzystanych technologii z uwzględnieniem ich kluczowych z punktu widzenia niniejszej pracy funkcjonalności.

3.2.1. Scala

Do stworzenia platformy na potrzeby niniejszej pracy wykorzystano język Scala. Głównym aspektem przemawiającym na korzyść tego języka jest to, że do jego działania wystarczy maszyna wirtualna Javy. Jest to język obiektowy podobnie jak Java ale łączy również zalety języków funkcyjnych, które w ostatnim czasie stają się coraz bardziej popularne. Zamiarem twórców było stworzenie języka eleganckiego oraz zwięzłego syntaktycznie. Mimo, że Scala jest językiem dynamicznie typowanym zapewnia tzw.



Rysunek 3.1: Relacja między kilkoma pojęciami identyfikującymi aktorów w systemie [1].

type safety [33]. Wprowadza również wiele innowacji oraz ciekawych rozwiązań do konstrukcji języka jak np. case classes, currying, zagnieżdżanie funkcji, DSL, tail recursion, słowo kluczowe lazy lub trait, konsktrukcja podobna do interfejsu z języka Java ale mogąca posiadać częściową implementację. Scala preferuje obiekty immutable. Obsługuje również funkcje wyższego rzędu oraz pozwala zwięźle definiować funkcje anonimowe, kładzie również duży nacisk na skalowalność. Dodatkową zaletą tego języka jest to, iż jest on w pełni kompatybilny z językiem Java, co oznacza, że możemy w nim używać bibliotek lub frameworków napisanych w Javie bez żadnych dodatkowych deklaracji. Ostatecznie program napisany w Scali jest kompilowany do kodu bajtowego Javy [36, 39, 15, 49]. Według autora niniejszej pracy do wad tego języka można zaliczyć m.in. to, że niektóre instrukcje da się wyrazić na wiele różnych sposobów, co utrudnia czytelność kodu oraz zwiększa trudność nauki tej technologii.

3.2.2. Akka

Jednym z kluczowych frameworków wykorzystanych w niniejszej pracy jest Akka. Akka jest projektem Open Source na licencji Apache 2. Posiada wersję przeznaczoną dla języka Java jak również dla języka Scala. Projekt Akka mimo stosunkowo niedługiej obecności na rynku jest w pełni gotowy do zastosowań produkcyjnych. Cechuje się obecnością wielu interesujących z punktu widzenia niniejszej pracy rozszerzeń, które zostały opisane w jednym z następnych podrodziałów, wyczerpującej dokumentacji oraz dużej i aktywnej społeczności rozwijającej ten produkt a także wsparciem komercyjnych firm. Akka używa modelu aktorowego aby zwiększyć poziom abstrakcji i oddzielić logikę biznesową od niskopoziomowego zarządzania wątkami oraz operacjami I/O [45, 14].

Podstawowym bytem w technologii Akka są aktorzy. Aktor zapewnia wysokopoziomową abstrakcję dla lepszej współbieżności oraz zrównoleglenia operacji. Jest on lekkim, wydajnym oraz sterowanym zdarzeniami procesem, który komunikuje się z innymi aktorami za pomocą asynchronicznych wiadomości przechowywanych w skrzynkach odbiorczych. Aktor enkapsuluje pewien stan i zachowania, które realizują określone zadania. Więcej informacji na temat koncepcji aktorów w projekcie Akka można znaleść w pozycji [14].

Każdy aktor w systemie może być identyfikowany na kilka różnych sposobów. Różnicę pomiędzy kilkoma z nich przedstawiono na rysunku 3.1. Więcej informacji na ten temat można znaleść w pozycji [1].

Aktorzy działają w obrębie systemu zwanego Actor System opisanego w pozycji [2]. System aktorowy można traktować jako pewną strukturę z zaalokowaną pulą wątków, stworzoną w ramach jednej logicznej aplikacji. Konfiguracją puli wątków zajmuje się Akka. Pewne ustawienia mogą być zmienione w pliku *application.conf*, który jest głównym plikiem konfiguracyjnym systemu aktorowego. System

aktorowy zarządza dostępnymi zasobami i może mieć uruchomione miliony aktorów gdyż instancja każdego z aktorów zajmuje zaledwie okolo 300 bajtów pamięci.

Akka framework dostarcza wszystkich zalet programowania reaktywnego [26] oraz zapewnia między innymi:

- współbieżność, dzięki zaadaptowaniu modelu aktorowego, programista może zatem skupić się na logice biznesowej zamiast zajmować się problemami współbieżności,
- skalowalność, asynchroniczna komunikacja pomiędzy aktorami dobrze skaluje się w systemach multiprocesorowych,
- odporność na błędy, framework Akka zapożyczył podejście z języka Erlang, co pozwoliło wykorzystać model *let it crash* [37] zwany też *fail fast* do zapewnienia sprawnego działania systemu i skrócenia jego niedostępności,
- architekture sterowaną zdarzeniami,
- transakcyjność,
- ujednolicony model programowania dla potrzeb wielowątkowości oraz obliczeń rozproszonych,
- Akka wspiera zarówno API języka Java jak i języka Scala [45].

Zastosowania projektu Akka są bardzo szerokie, można je odnaleść chociażby w następujących dziedzinach:

- analiza danych,
- bankowość inwestycyjna,
- eCommerce,
- symulacje,
- media społecznościowe.

Systemy w których potrzebujemy uzyskać wysoką przepustowość oraz małe opóźnienia są dobrym kandydatem do wykorzystania frameworka Akka [14].

3.2.3. Rozszerzenia Akka

W poniższym rodziale opisano kilka ciekawych z puntku widzenia niniejszej pracy rozszerzeń projektu Akka wykorzystanych w implementancji stworzonego rozwiązania.

Akka Cluster

Jest to najbardziej istotne rozszerzenie z punktu widzenia niniejszej pracy. Została na nim oparta komunikacja pomiędzy urządzeniami działającymi w obrębie klastra. Zapewnia ono pewien poziom abstrakcji dla protokołu TCP/IP [8].

Dostarcza również odporny na awarię oraz zdecentralizowany serwis członkostwa (ang. membership service) oparty na protokole Gossip [24] oraz automatycznej detekcji niedziałających węzłów (ang. failure detector).

Pojęcia:

- węzeł (ang. node) - logiczny członek klastra, może istnieć wiele węzłów na jednej fizycznej maszynie, identyfikowany krotką: nazwa_hosta:port:uid,
- klaster (ang. cluster) - grupa węzłów zarejestrowana w serwisie członkostwa,

- lider (ang. leader) - węzeł odpowiedzialny za kluczowe akcje pozwalające zachować odpowiedni stan klastra w przypadku dołączania nowych lub awarii istniejących węzłów [14].

Lider jest tylko rolą jaką posiada dany węzeł, każdy węzeł może zostać liderem oraz każdy węzeł jest w stanie w sposób deterministyczny wyznaczyć lidera. Węzły mogą też posiadać inne role, które mogą się przydać np. w ograniczeniu zasięgu komunikacji.

Protokół Gossip wspomniany powyżej pozwala rozpropagować stan klastra do wszystkich jego węzłów, tak aby każdy węzeł miał takie same informację o pozostałych członkach klastra. Protokół ten pomaga uzyskać zbieżność stanu klastra we wszystkich jego węzłach w skończonym czasie.

Czlonkostwo w klastrze jest rozpoczynane komendą *join* wysyłaną do jednego z aktywnych członków klastra, gdy każdy węzeł klastra otrzyma informację o dołączającym węźle dzięki protokołowi Gossip, taki węzeł może zostać uznany za osiągalny. Może on jednak utracić ten stan gdy np. wystąpią jakieś problemy z siecią. Stan nieosiągalny może trwać jedynie określony czas po upływie którego jeśli węzeł nie powróci do pełnej sprawności traci on status członka klastra. Każdy węzeł poza nazwą hosta oraz portem jest identyfikowany dodatkowo unikalnym identyfikatorem, tzw. uid, co pozwala na uruchomienie kilku węzłów klastra na jednej fizycznej maszynie. Jeżeli członek klastra ulegnie awarii lub opuści klaster na własne życzenie nie może on ponownie dołączyć do klastra dopóki system aktorowy uruchomiony na nim nie zostanie zrestartowany, co pozwoli na wygenerowanie nowego uid. Każdy węzeł może zmienić swój stan członkostwa lub może on zostać zmieniony automatycznie dzięki automatycznej detekcji niedziałających węzłów. Ustawienia detekcji niedziałających węzłów oraz protokołu Gossip są konfigurowalne, tzn. że można np. ustalić czas po którym węzeł zostanie uznany za nieosiągalny. Zdarzenia związane ze zmianą stanu węzłów klastra mogą być subskrybowanie przez istniejących członków, co ułatwia implementację monitoringu stanu klastra. Więcej szczegółów na temat idei członkostwa w klastrze oraz dostępnych stanów węzłów można znaleść w pozycjach [14, 48].

Kolejną ciekawą opcją w rozszerzeniu Akka Cluster jest wykorzystanie bibliotek Sigar [30] oraz integracja z nimi. Pozwalają one na dostęp do informacji systemowych takich jak zużycie CPU, wykorzystanie pamięci RAM, stan sieci. Mogą zostać wykorzystane do implementacji load balancigu lub monitorowania obciążenia urządzeń klastra [4, 3].

Cluster Singleton

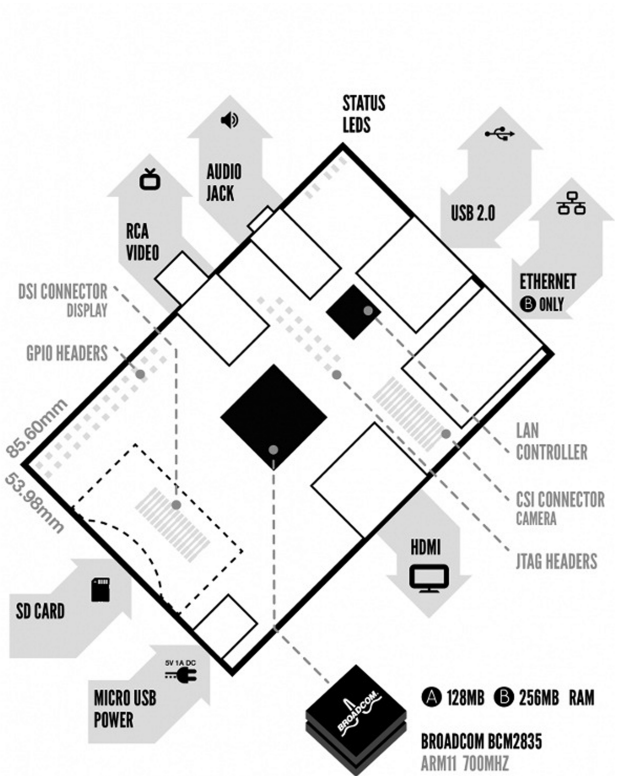
Cluster Singleton jest rozszerzeniem projektu Akka zapewniającym jedną instancję aktora danego typu w obrębie klastra lub w obrębie węzłów z wybraną rolą. Zostało ono wykorzystane do zaimplementowania modelu master/slave opisanego w poprzednim podrozdziale. Rozszerzenie to dostarcza implementacji menedżera, który pozwala zarządzać instancjonowaniem aktora-singletona. Dostęp do działającej instancji jest możliwy z każdego urządzenia działającego w klastrze i odbywa się przez aktora pośrednika, tzw. proxy [17].

Distributed Publish Subscribe in Cluster

Kolejne rozszerzenie projektu Akka, które umożliwia komunikację między aktorami bez posiadania informacji na których konkretnie urządzeniach poszczególni aktorzy są uruchomieni, czyli lokacja aktora jest transparentna z punktu widzenia komunikacji. Dostarcza ono aktora-mediatora, który zarządza rejestracją innych aktorów do konkretnych kanałów komunikacji którymi są zainteresowani. Zachowanie to, można porównać z subskrypcją RSS [55]. Wiadomość opublikowana w danym kanale powinna zostać dostarczona do wszystkich aktorów, którzy zostali w nim uprzednio zarejestrowani. Rozszerzenie to pozwala również wysłać wiadomość do jednego lub większej ilości aktorów pasujących do określonego wzorca. W niniejszej pracy zostało ono wykorzystane do przesyłania wyników zadania do klienta oraz do realizacji migracji populacji w implementacji algorytmów genetycznych [11].

Akka Persistence

Rozszerzenie Akka Persistence pozwala na zapisywanie stanu aktorów w bazie danych. Pozwala to na odtworzenie stanu aplikacji w przypadku awarii i zapewnia w pewnym stopniu niezawodność dzi-



Rysunek 3.2: Schemat Raspberry Pi model B [[18]].

ałania systemu [6]. Dostępnych jest wiele różnych dodatków dedykowanych dla różnych technologii bazodanowych. W niniejszej pracy wykorzystano dodatek przeznaczony do pracy z bazą MongoDB [7].

3.2.4. Raspberry Pi

Do testów platformy stworzonej na potrzeby niniejszej pracy wykorzystano jedno z urządzeń SoC [31] jakim jest Raspberry Pi model B. Jest to urządzenie stworzone przez fundację non-profit oparte na architekturze ARM.

Na rysunku 3.2 przedstawiono schemat urządzenia.

Urządzenie posiada jednostkę CPU oparty o taktowaniu 700MHz oraz 512MB pamięci RAM. Poza procesorem CPU jest również wyposażone m.in. w jednostkę GPU, dwa porty USB, złącze kart SD (na których może być zainstalowana dystrybucja Linuxa), złącze Ethernet 100Mb/s oraz złącze MicroUSB wykorzystane do zasilania. Pełną specyfikację urządzenia można znaleść w pozycji [18].

3.2.5. MongoDB

MongoDB jest bazą NoSQL opartą na dokumentach typu JSON, co pozwala na elastyczność oraz przejrzystość w przechowywaniu danych. Jest to technologia Open Source. Jako, że jest to baza NoSQL posiada ona elastyczny schemat danych. Wspiera replikację oraz sharding a także map reduce. Technologia ta została wykorzystana ze względu na jej łatwe użycie, dobrą skalowalność oraz wydajność przy wykorzystaniu stosunkowo niewielkich zasobów obliczeniowych [13].

Kolejnymi argumentami przemawiającymi na korzyść tej technologii są łatwa dostępność dokumentacji oraz integracja z innymi technologiami wykorzystanymi w niniejszej pracy.

3.2.6. Play Framework i AngularJS

Technologie Play Framework oraz AngularJS zostały wykorzystane do przygotowania aplikacji webowej będącej interfejsem klienta dzięki któremu może sterować on działaniem platformy, monitorować ją oraz oglądać wyniki obliczeń.

Play Framework jest to technologia server-side, działająca zarówno z językiem Java jak i Scala. Integruje ona komponenty oraz API potrzebne do stworzenia aplikacji webowej opartej o wzorzec MVC. Bazuje on na lekkiej, bezstanowej architekturze, niskim zużyciu zasobów, wysokiej skalowalności oraz programowaniu reaktywnym. Do komunikacji wykorzystuje metody protokołu HTTP oraz pozwala na implementację serwisów w technologii REST. Play zapewnia również wsparcie dla technologii WebSocket wykorzystanej w niniejszej pracy oraz łatwą integrację z frameworkiem Akka [16, 53].

Z kolei AngularJS jest to technologia stworzona z użyciem języka JavaScript, ze wsparciem komercyjnych firm takich jak Google. Rozszerza możliwości języka HTML oraz CSS a ponadto pozwala tworzyć tzw. SPA (ang. Single Page Applications). W niniejszej pracy wykorzystano ją do prezentowania informacji odebranych z aplikacji serwerowej oraz interakcji z użytkownikiem za pomocą przeglądarki internetowej [12, 43].

3.2.7. Wykorzystane szablony

Platforma Typesafe w której skład wchodzą technologie wykorzystane w niniejszej pracy, takie jak Play Framework, Akka oraz Scala dostarcza bardzo wyczerpującą dokumentację wraz z wieloma przykładami oraz szablonami na licencji Public Domain.

Rozwiązanie stworzone na potrzeby niniejszej pracy bazowało początkowo na dwóch szablonach, pierwszy zawierał przykład monitoringu urządzeń podłączonych do klastra natomiast drugi dystrybucję zadań pomiędzy urządzenia [23, 32].

3.3. Platforma

Niniejszy podrozdział omawia wybrane aspekty stworzonej platformy, takie jak architektura, protokoły komunikacji oraz przydzielanie zadań.

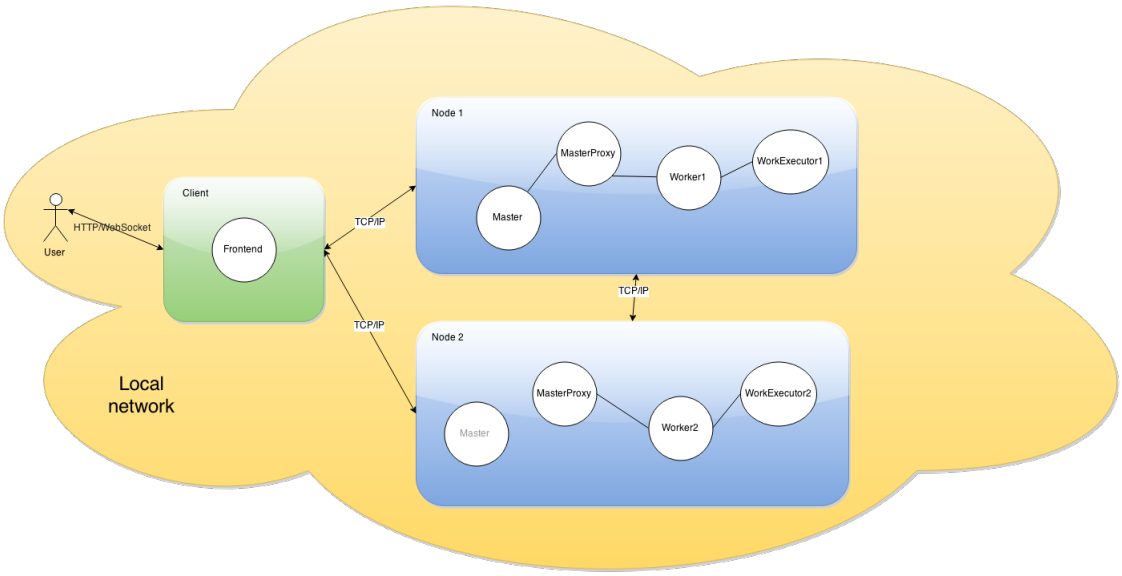
3.3.1. Architektura

Węzeł klastra jest to instancja aplikacji niezwiązana z fizycznym urządzeniem. Co oznacza, że na jednym urządzeniu może być uruchomionych kilka węzłów, czyli kilka aplikacji. W skład platformy wchodzą dwie aplikacje nazwane *frontend* i *backend*. Posiadają one różne odpowiedzialności jak i rodzaje uruchomionych na nich aktorów o których zostanie wspomniane w kolejnym podrozdziale.

Aplikacja *frontend* odpowiada za interakcje z użytkownikiem dzięki takim technologiom jak Play Framework, AngularJS, WebSocket oraz protokołowi HTTP. Komunikuje się ona z aplikacją *backend* w celu zlecenia wykonania zadania oraz odebrania wyników. Posiada ona interfejs webowy. Instancja aplikacji jest traktowana jako osobny węzeł klastra.

Aplikacja *backend* nie posiada interfejsu użytkownika. Zajmuje się realizacją zleconych zadań. Instancja tej aplikacji odpowiada jednemu węzłowi klastra. W klastrze może być uruchomionych wiele instancji, po jednej na każdym urządzeniu lub jeśli urządzenie posiada wystarczająco mocy obliczeniowej może posiadać kilka uruchomionych aplikacji. Każda uruchomiona aplikacja posiada uruchomionego workera oraz tylko jedna z nich posiada również uruchomionego mastera, gdyż w klastrze powinna być tylko jedna aktywna instancja mastera.

Na rysunku 3.3 przedstawiono podgląd architektury, zielonym kolorem zaznaczono aplikację *frontend* natomiast niebieskim aplikację *backend*. Założono, że każda aplikacja jest uruchomiona na osobnym urządzeniu pracującym w klastrze, są dwa urządzenia wykonujące obliczenia oraz jedno urządzenie klienckie. Na wspomnianym rysunku pokazano także kilka aktorów uruchomionych na wybranych urządzeniach. Założono również, że urządzenia pracują w sieci lokalnej z wykorzystaniem technologii



Rysunek 3.3: Podgląd architektury.

Ethernet. Użytkownik komunikuje się za pomocą protokołu HTTP z urządzeniem klienckim, które z kolei przesyła odpowiednie żadania do urządzenia posiadającego aktywnego mastera. Urządzenie klienckie nie musi posiadać informacji o tym gdzie dokładnie jest aktywny master, ponieważ dostęp do mastera odbywa się poprzez proxy. Natomiast aplikacja kliencka powinna mieć podany adres IP przynajmniej jednego urządzenia pracującego w klastrze. Może mieć informację o kilku adresach IP urządzeń pracujących w klastrze, co pozwala uniknąć problemów gdy jedno z urządzeń nie odpowiada. Najłatwiejszym sposobem podania adresów urządzeń pracujących w klastrze jest użycie dodatkowych argumentów podczas uruchamiania aplikacji. Szczegóły tych ustawień oraz sposobu uruchamiania aplikacji wchodzących w skład przygotowanej platformy zostały opisane w dodatku A załączonym do niniejszej pracy.

Urządzenia wykonujące obliczenia w klastrze komunikują się ze sobą za pomocą protokołu TCP/IP, którym zarządza Akka. Odkąd aktywny master posiada listę wszystkich zarejestrowanych workerów potrafi on w łatwy sposób nawiązywać z nimi kontakt oraz informować ich o nadchodzących zadaniach od klienta. Z kolei komunikacja w drugą stronę, np. gdy worker skończył wykonywać zadanie i chce wysłać zgromadzone wyniki odbywa się poprzez proxy, ponieważ worker nie wie na którym urządzeniu znajduje się aktywny master. Gdy master otrzyma wyniki zadania od realizującego je workera odsyła je do klienta za pomocą rozszerzenia *Distributed Publish Subscribe in Cluster* opisanego w jednym z poprzednich podrozdzialow. Co oznacza, że nie musi posiadać informacji o adresie IP klienta, ponieważ publikuje wiadomości w wybranym kanale i otrzymają je tylko zainteresowani klienci, którzy obserwują dany kanał.

Na rysunku 3.3 pokazano jedynie kilka najważniejszych aktorów działających na urządzeniach będących częścią klastra. Zostali oni opisani razem z pozostałymi, którzy również są aktywni podczas działania platformy w następnym podrozdziale wraz z podziałem na aplikacje w których są wykorzystywani.

3.3.2. Wybrani aktorzy

Aplikacja *backend*:

- ClusterSingletonManager, aktor dostarczony wraz z rozszerzeniem *Akka Cluster Singleton*, który odpowiada za to aby w klastrze była aktywna tylko jedna instancja mastera. Zostaje on uruchomiony na wszystkich węzłach lub na węzłach z wybraną rolą. Instancjonuje aktora singletona

na najstarszym węźle, czyli takim, który dołączył do klastra najwcześniej. Monitoruje on również stan dostępności węzła na którym uruchomiony jest aktor singleton i w przypadku awarii startuje nową instancję na innym węźle.

- ClusterSingletonProxy, aktor współpracujący z poprzednim opisanym aktorem i będący proxy zapewniającym dostęp do aktora singletona czyli w tym przypadku mastera. Jest on uruchomiony na każdym węźle, który potrzebuje komunikować się z masterem i działa jak router przekierowujący wszystkie przychodzące wiadomości do aktora singletona. Jeśli aktor ten jest niedostępny np. ze względu na awarię lub jest w trakcie odzyskiwania sprawności, zadaniem aktora ClusterSingletonProxy jest przechowywanie wszystkich odebranych wiadomości aż do momentu ponownego nawiązania komunikacji [17].

- DistributedPubSubMediator, aktor ten zostaje uruchomiony na wszystkich węzłach klastra lub na węzłach z wybraną rolą. Zarządza rejestracją innych aktorów do wybranych kanałów komunikacji oraz replikuje tą wiedzę pośród inne instancje pracujące na pozostałych urządzeniach klastra tak aby zachować spójność tych danych w obrębie klastra. Zajmuje się również wysyłaniem wiadomości z każdego urządzenia klastra do zarejestrowanych aktorów pracujących na jakimkolwiek urządzeniu w obrębie klastra [11].

- Master, jest to tzw. aktor singleton. Oznacza to, że w obrębie klastra powinna być aktywna tylko jedna instancja tego aktora. Jest to zapewnione poprzez aktora ClusterSingletonManager opisanego powyżej. Dostęp do mastera uzyskuje się poprzez proxy, czyli aktora ClusterSingletonProxy z każdego urządzenia pracującego w klastrze. Głównymi zadaniami tego aktora są: przydzielanie zadań workerom, zarządzanie ich rejestracją, odbieranie wyników i przesyłanie ich do klienta, reagowanie w sytuacji awarii workera oraz zapisywanie wyników do bazy.

- Worker, jest to aktor realizujący zadania otrzymane od mastera po uprzedniej rejestracji. Każdy worker uruchamia jedną instancję aktora wykonującego właściwe obliczenia, po to aby zachować responsywność w komunikacji z masterem. Zajmuje się on również procesem migracji, gdzie znowu jest wykorzystywany mechanizm publikowania w wybranym kanale dostarczony wraz z rozszerzeniem *Distributed Publish Subscribe.*

- WorkExecutor, aktor wykonujący właściwe obliczenia czyli realizujący zadanie zlecone przez klienta. Jest on instancjonowany przez workera i komunikuje się tylko z nim.

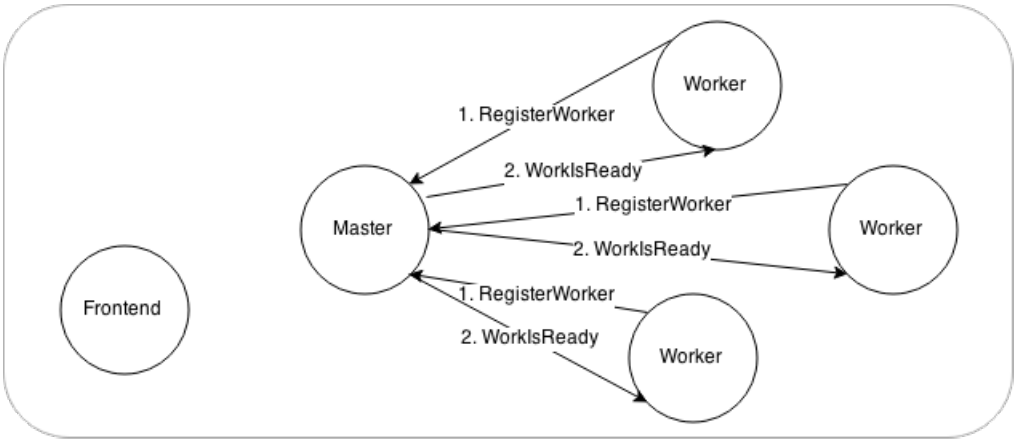
Aplikacja *frontend*:

- Frontend, aktor z którym komunikuje się użytkownik. Odbiera on wyniki zadania od mastera oraz przesyła mu zadania do wykonania. Do komunikacji z masterem używa proxy, natomiast wyniki odbiera dzięki rozszerzeniu *Distributed Publish Subscribe* oraz rejestracji w wybranym kanale komunikacji.

- Metrics, aktor subskrybujący zdarzenia związane ze zmianą różnych metryk klastra, np. obciążenia CPU lub pamięci RAM. Dzięki połączeniu WebSocket przesyła te informację na widok.

- Monitor, aktor obserwujący zdarzenia związane z członkostwem w klastrze. Jeśli jakiś węzeł dołącza do klastra lub go opuszcza taka informacja jest aktualizowana na widoku.

Poza wyżej wymienionymi aktorami działającymi w aplikacji *frontend*, posiada ona również kilka tych samych aktorów co aplikacja *backend*, m.in. ClusterSingletonProxy oraz DistributedPubSubMediator, którzy służą do komunikacji z masterem oraz odbieraniem wyników realizowanych zadań.



Rysunek 3.4: Rejestracja Workerów.

Tablica 3.1: Opis komunikatów - rejestracja workerów.

Nazwa komunikatu	Opis
RegisterWorker	Komunikat rejestrujący workera o danym ID w serwisie mastera.
WorkIsReady	Informacja o tym, że master posiada zakolejkowane zadanie po które dany worker może się zgłosić.

3.3.3. Protokoły komunikacji

Na rysunkach 3.4 oraz 3.5 przedstawiono dwa najważniejsze protokoły definiujące kolejność oraz rodzaje przesyłanych wiadomości służących do komunikacji na linii Frontend - Master - Worker - WorkExecutor. Widoczni są tam również aktorzy biorący lub nie udział we wspomnianej komunikacji.

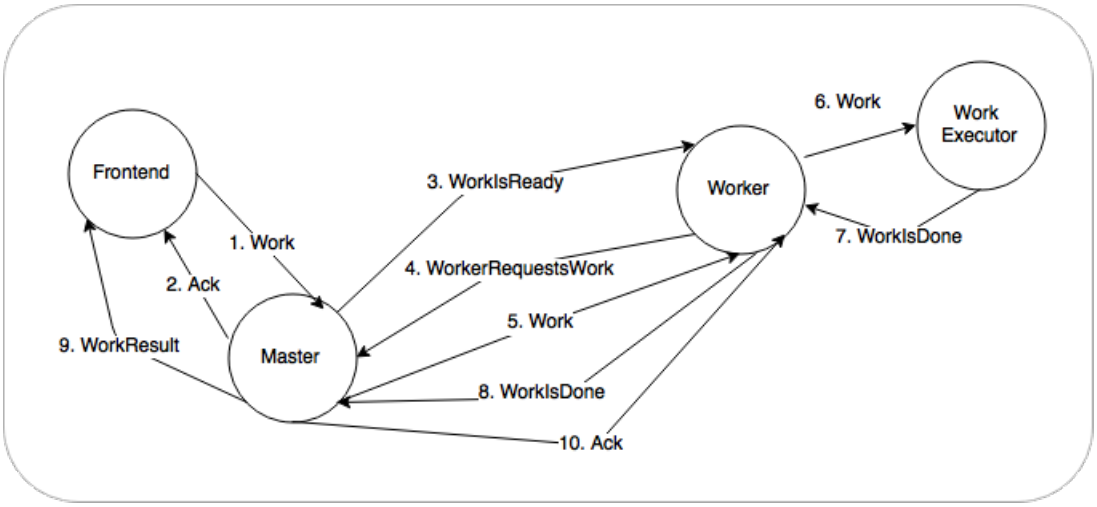
Na rysunku 3.4 przedstawiono proces rejestracji workerów w serwisie mastera, po to aby master mógł ich poinformować o tym, że posiada jakieś zadanie, które trzeba zrealizować.

W tabeli 3.1 przedstawiono opis komunikatów zawartych na rysunku 3.4.

Na rysunku 3.5 pokazano przepływ informacji podczas zlecania oraz realizacji zadania. Dla uproszczenia przedstawiono przypadek zadania zrealizowanego bez przesyłania częściowych wyników. W przypadku zadań trwających długo lub gdzie czas trwania nie jest z góry określony tzw. *long-running jobs* wprowadzono również komunikat *WorkInProgress*, który jest przesyłany cyklicznie w trakcie wykonywania zadania i zawiera jego częściowe wyniki, można go umiejscowić pomiędzy komunikatami *Work* a *WorkIsDone*.

W tabeli 3.2 przedstawiono opis komunikatów przesyłanych w trakcie procesu realizacji zadań.

Do odpowiedniego rozłożenia obciążenia pomiędzy pracujących workerów początkowo rozważano podejście gdzie to master decyduje, któremu workerowi przydzielić zadanie bazując między innymi na metrykach takich jak obciążenie CPU lub zajętość pamięci RAM. Jednak okazało się ono nieefektywne w implementacji testowanych algorytmów genetycznych, ponieważ te metryki zmieniały zazwyczaj z opóźnieniem pozostawiając workera w stanie bezczynności przez jakiś określony czas. Zdecydowano się zatem zaimplementować podejście zwane *Work Pulling Pattern* [34] gdzie to wolny worker zgłasza się po zadanie a nie oczekuje na reakcję mastera.



Rysunek 3.5: Protokół Master/Worker.

Tablica 3.2: Opis komunikatów przesyłanych w trakcie realizacji zadania.

Nazwa komunikatu	Opis
Work	Komunikat przechowujący parametry konfiguracyjne zadania jak również jego ID.
Ack	Potwierdzenie odebrania poprzedniego komunikatu.
WorkIsReady	Informacja o tym, że master posiada zakolejkowane zadanie po które dany worker może się zgłosić.
WorkerRequestsWork	Komunikat przechowujący ID workera i informujący mastera, że dany worker jest wolny i może zająć się kolejnym zadaniem.
WorkIsProgress	Komunikat przechowujący częściowe wyniki zadania jak również jego ID oraz ID workera.
WorkIsDone	Komunikat przechowujący WorkResult oraz ID workera.
WorkResult	Komunikat przechowujący końcowe wyniki zadania jak i jego ID.

4. Zastosowanie w praktyce

Rozdział ten przedstawia sposoby wykorzystania rozwiązania stworzonego na potrzeby niniejszej pracy, jego możliwości oraz wyniki testów.

4.1. Użycie

W niniejszym podrozdziale opisano wykorzystanie stworzonej platformy w praktyce, sposób jej użycia oraz zasady działania a także wyniki zaimplementowanych rozwiązań.

4.1.1. Monitoring

Na rysunku 4.1 przedstawiono interfejs klienta prezentujący obecny stan klastra. Wylistowano na nim również listę urządzeń pracujących w klastrze, z uwględnieniem aktywnego mastera, adresu IP wraz z portami, dostępność urządzenia oraz kilka metryk, m.in. zużycie CPU oraz pamięci RAM.

Stan klastra propagowany jest za pomocą protokołu Gossip opisanego w poprzednim rozdziale. Jeżeli jakieś urządzenie nie odpowiada przez określony czas w wyniku np. problemów z siecią, jest ono uznawane jako niedostępne, czyli nie może już przyjmować żadnych zadań ale może jeszcze powrócić do pracy w klastrze jeżeli odzyska sprawność w ciągu najbliższych kilku sekund. Jeżeli nie, w takim przypadku jest ono usuwane z listy urządzeń pracujących w klastrze. Po usunięciu urządzenia z klastra nie może ono dołączyć ponownie do klastra dopóki aplikacja uruchomiona na tym urządzeniu nie zostanie zrestartowana. Pozwala to zapobiec sytuacji, gdy w trakcie problemów z siecią, dane urządzenie odłączy się na chwilę od klastra aktywując własnego mastera a następnie po powrocie do klastra bedą aktywne dwa mastery. Dzieje się tak dlatego, że w przypadku gdy urządzenie utraci kontakt z pozostałymi urządzeniami, nie może ono z pewnością stwierdzić co było przyczyną awarii i zakłada, że jest jedynym urządzeniemi w klastrze.

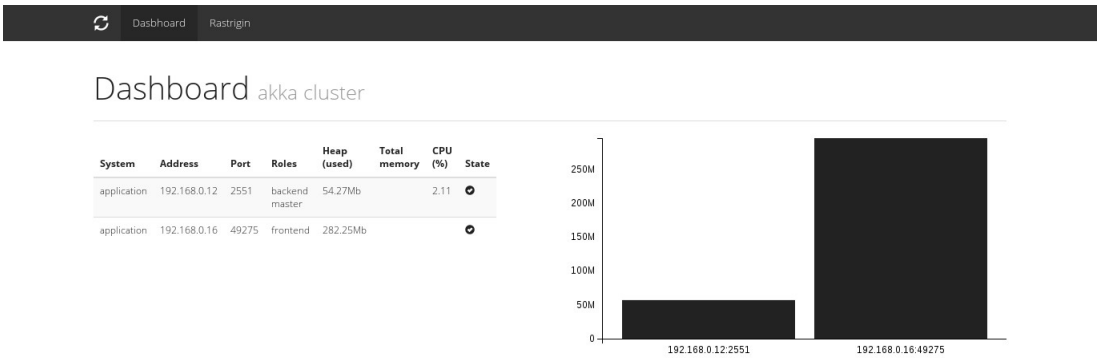
W trakcie awarii urządzenia podczas wykonywania obliczeń dzięki występowaniu migracji opisanej w jednym z kolejnych podrozdziałów nie tracimy wszystkich wyników pracy a jedynie te uzyskane od poprzedniej migracji. Stan mastera jest zapisywany w bazie danych, co pozwala na jego odtworzenie w przypadku awarii urządzenia z aktywnym masterem. Do bazy danych persystowane są zserializowane zdarzenia zgodnie ze wzorcem Event Sourcing opisanym w pozycjach [19, 20]. Zdarzenia te odzwierciedlają stan przyjętego zadania. Listę zdarzeń oraz ich opis przedstawiono w tabeli 4.1.

Dla przykładu, jeżeli klient zadał zadanie i zmieniło ono stan na WorkAccepted (czyli zostało zaakceptowane ale nie przesłane jeszcze do żadnego workera) a w tym samym czasie nastąpiła jakaś awaria przerywająca pracę mastera, kolejna instancja mastera zostanie aktywowana na innym urządzeniu i odtworzy stan mastera, który uległ awarii, co oznacza, że klient nie będzie musiał ponownie rozpoczynać zadania, lecz jego poprzednio rozpoczęte zadanie zostanie przekazane do realizacji do wolnego workera.

Diagram prezentujący dopuszczalne zmiany stanów zadania przedstawiono na rysunku 4.2.

4.1.2. Implementacja algorytmów genetycznych

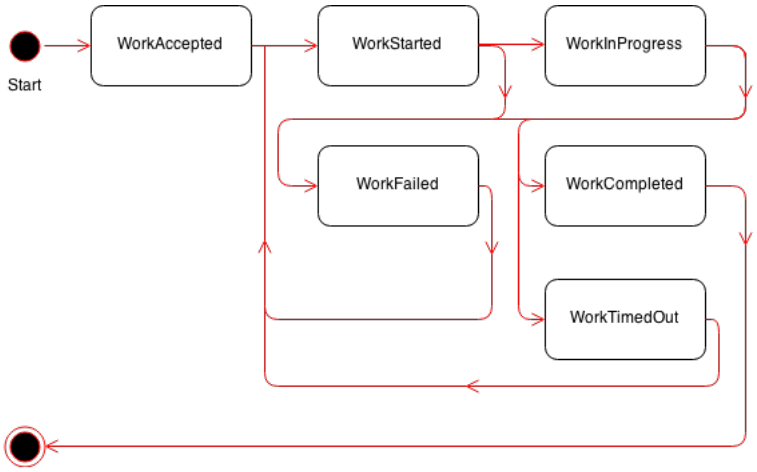
Rozwiązanie stworzone na potrzeby niniejszej pracy przetestowano dzięki zaimplementowaniu algorytmów genetycznych w odmianie wyspowej opisanych w poprzednim rozdziale.



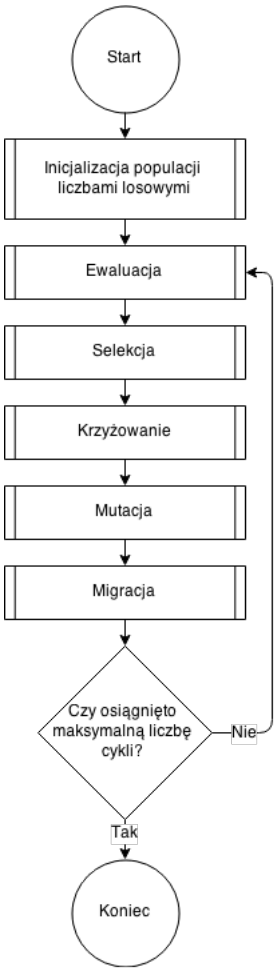
Rysunek 4.1: Podgląd stanu klastra.

Tablica 4.1: Lista zdarzeń sterujących stanem zadania.

Nazwa stanu	Opis
WorkAccepted	Stan zadania po przyjęciu go przez mastera oraz wysłaniu potwierdzenia do klienta.
WorkStarted	Stan zadania przekazanego do realizacji do workera.
WorkInProgress	Stan zadania w trakcie trwania obliczeń, przechowujący częściowe wyniki zadania.
WorkCompleted	Stan zadania otrzymany po zakończeniu obliczeń, przechowujący końcowe wyniki.
WorkerFailed	Stan zadania zakończonego niepowodzeniem.
WorkerTimedOut	Stan zadania ustawiany jeśli worker pracujący nad danym zadaniem nie odeśle wyników w określonym czasie.



Rysunek 4.2: Przejścia pomiędzy stanami zadania.



Rysunek 4.3: Schemat blokowy ewolucji.

Implementacja ta, została oparta na implementacji algorytmów genetycznych przedstawionej w książce Scala for Machine Learning [51] oraz dostosowana do optymalizacji funkcji ciągłej.

W przypadku problemu Rastrigina gen jest liczbą typu Double, chromosomy zawierają listę genów czyli obrazują punkty w przestrzeni rozwiązań. W przypadku dwuwymiarowej funkcji Rastrigina jest to lista dwuelementowa. Natomiast populacja zawiera listę chromosomów.

Na rysunku 4.3 przedstawiono schemat blokowy ewolucji.

Zadanie jest konfigurowane parametrami przedstawionymi w tabeli 4.2.

Migracja odbywa się poprzez wybranie pewnej ilości najlepszych osobników populacji oraz wysłanie ich do innego węzła klastra. Wykonuje się ona co określoną liczbę cykli konfigurowalną w parametrach zadania. Do wyboru są dwa rodzaje selekcji węzła do którego wysyłani są migrujący osobnicy. Pierwszy to jest wybór losowy a drugi metodą round-robin, czyli wysyłanie migrującej części populacji na zmianę do każdego kolejnego węzła [27]. Migracja odbywa się bez użycia mastera, do jej działania wykorzystano rozszerzenie frameworka Akka o nazwie *Distributed Publish Subscribe in Cluster* opisane w poprzednim rozdziale.

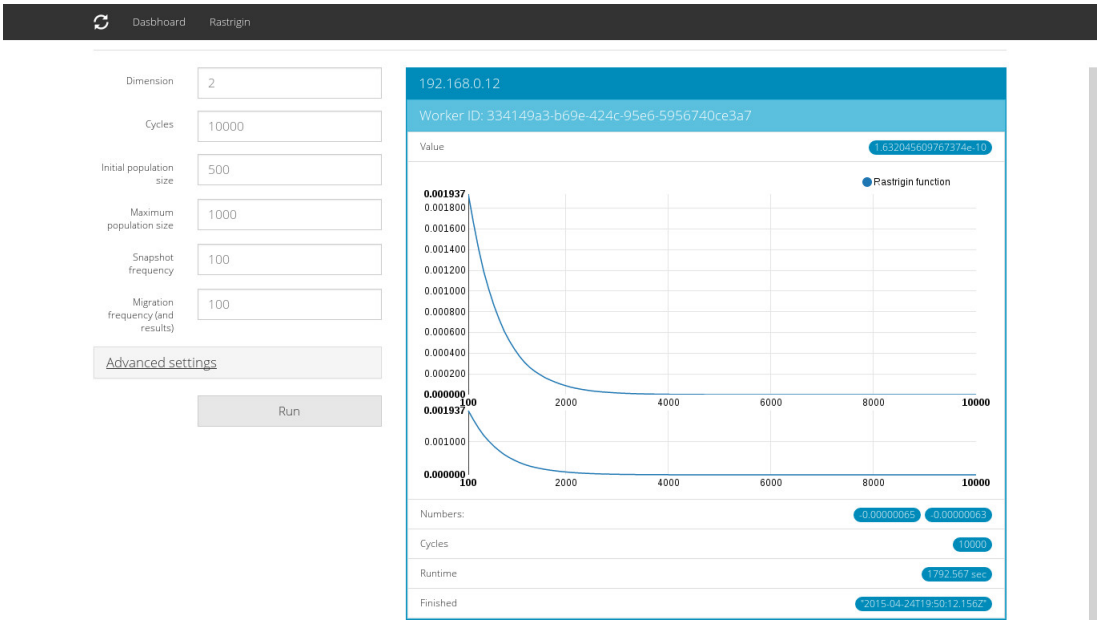
Inicjalizacja populacji odbywa się w każdym węźle klastra osobno. W przypadku problemu Rastrigina opisanego w kolejnym podrozdziale, początkowa populacja składa się z osobników zainicjalizowanych losowymi liczbami z przedziału $[-5.12, 5.12]$.

Na rysunku 4.4 zaprezentowano wygląd interfejsu użytkownika pozwalającego na podgląd wyników rozpoczętych zadań oraz rozpoczęcie nowych. Widoczne jest tutaj każde z urządzeń wykonujących obliczenia w klastrze i są one identyfikowane po adresie IP oraz ID workera.

Tablica 4.2: Parametry zadania.

Nazwa parametru	Opis
Dimension	Wymiar optymalizowanej funkcji ciągłej.
Cycles	Maksymalna ilość cykli ewolucji.
Initial population size	Początkowy rozmiar populacji (podczas krzyżowania i mutacji liczba osobników populacji zostaje wzrasta).
Maximum population size	Maksymalny rozmiar populacji.
Snapshot frequency	Częstotliwość persystowania wyników obliczeń (w tym przypadku osobników populacji). *
Migration frequency	Częstotliwość występowania migracji. *
Mutation parameter	Parametr mutacji, określa wpływ mutacji na geny.
Migration factor	Procent migrującej populacji.

* częstotliwość jest określana przez ilość cykli



Rysunek 4.4: Problem Rastrigina.

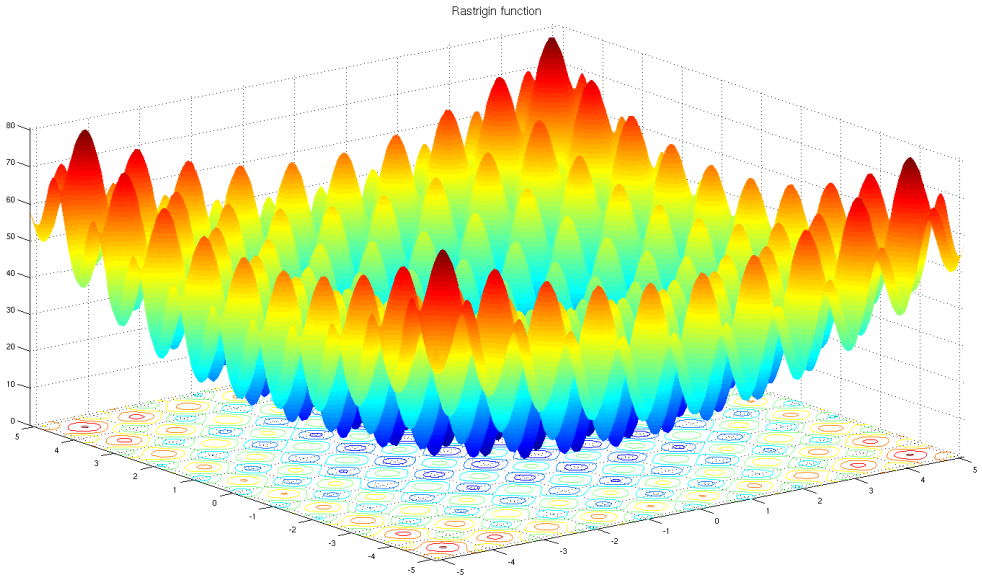
Wyniki prezentowane są w czasie rzeczywistym przy użyciu technologi WebSocket [25]. Dla każdego zadania rysowany jest wykres przedstawiający wartość funkcji Rastrigina zmieniającą się w trakcie trwania ewolucji, czyli w kolejnych cyklach algorytmu. Wykres narysowano za pomocą biblioteki D3.js [10].

4.1.3. Problem Rastrigina

Implementacja algorytmów genetycznych opisana w poprzednim podrozdziale została wykorzystana do znalezienia minimum funkcji Rastrigina.

Funkcja Rastrigina jest funkcją niewypukłą. Ze wglądu na to, że posiada wiele minimów lokalnych oraz jedno globalne dla $x = 0$, $f(x) = 0$ bywa trudna w optymalizacji, ponieważ algorytmy optymalizacyjne mogą utknąć w którymś z lokalnych minimów tracąc szanse na znalezienie minimum globalnego. Funckja ta często znajduje zatem zastosowanie jako funkcja testująca algorytmy optymalizacyjne. Została ona również użyta jako problem testowy w niniejszej implementacji.

Wzór funkcji Rastrigina przedstawiono poniżej



Rysunek 4.5: Wykres funckji Rastrigina 3D [21].

$$f(x) = An + \sum_{i=1}^{n} [x_i^2 - Acos(2\pi x_i)], \tag{4.1}$$

gdzie $A = 10$, $x_i \epsilon [-5.12, 5.12]$ a n oznacza wymiar funkcji [46].
Wykres funkcji Rastrigina przedstawiono na rysunkach 4.5 oraz 4.6.

4.1.4. Dystrybucja

Do łatwej dystrybucji platformy między różnymi urządzeniami wykorzystano dodatek projektu Akka o nazwie Microkernel oraz rozszerzenie Sbt o nazwie sbt-native-packager.

Akka Microkernel oraz sbt-native-packager dostarczają mechanizmu archiwizowania dzięki któremu można udostępniać aplikację jako pojedyńczy plik bez potrzeby uruchamiania lub instalowania dodatkowych aplikacji, dostarczania zależności lub ręcznego tworzenia skryptów startowych. Pozwalają również na wystartowanie systemu aktorowego używając klasy z metodą statyczną main [5, 28].

Umożliwia to łatwą instalację oraz uruchomienie platformy na różnych urządzeniach klastra. Do uruchomienia platformy na danym urządzeniu wystarczy aby posiadało ono zainstalowany system wraz z wirtualną maszyną Javy oraz archiwum z platformą, które po rozpakowaniu pozwala uruchomić aplikację za pomocą jednego skryptu wykonywalnego. Żadne dodatkowe oprogramowanie nie jest wymagane.

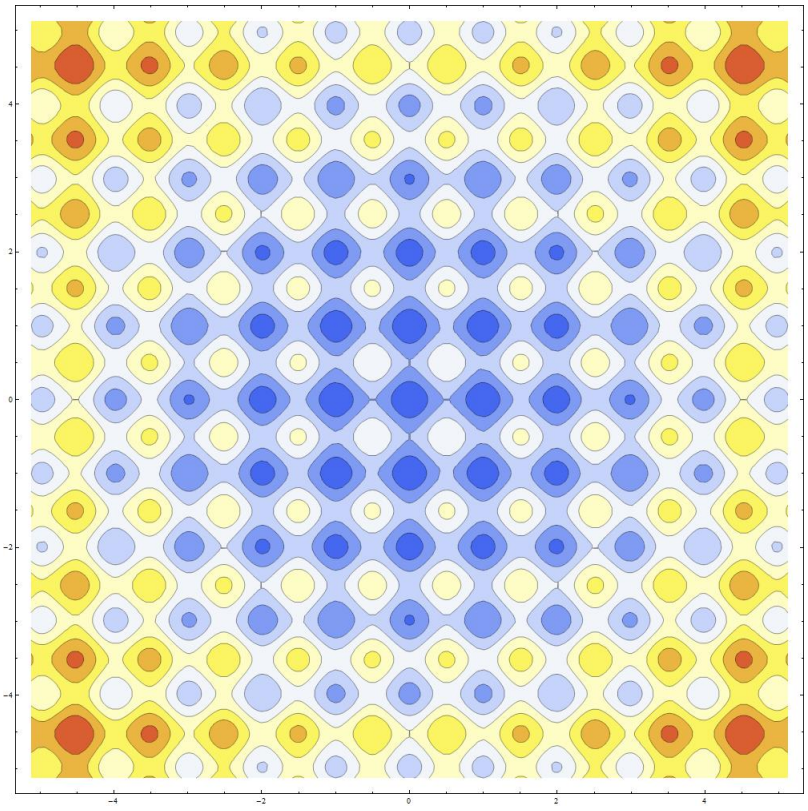
4.2. Wyniki testów

Do przeprowadzenia testów zostało wykorzystanych kilka urządzeń Raspberry Pi. Jako problem testowy wykorzystano optymalizację wielowymiarowej funkcji Rastrigina opisanej w poprzednim podrozdziale.

4.2.1. Wyniki obliczeń

Ze względu na ograniczoną precyzję obliczeń nie udało się uzyskać wyniku równego poszukiwanemu minumum, czyli $f(x) = 0$.

Najlepsze uzyskane wyniki oscylowały w okolicach $2e - 30$.



Rysunek 4.6: Wykres funkcji Rastrigina 2D [21].

Do obliczeń wykorzystano typ Double, który w języku Scala cechuje się podwójną precyzją, zapisaną na 64 bitach [49].

Na poniższych wykresach przedstawiono wyniki oraz czasy ich uzyskania w zależności od różnych ustawień początkowych zadania.

Rozważono tutaj problem dla jednego urządzenia w celu porównania go z bardziej istotnym przypadkiem dla celów niniejszej pracy a mianowicie wielu urządzeń pracujących w klastrze, który rozważono w następnym podrozdziale poświeconemu skalowalności rozwiązania.

Na rysunku 4.7 przedstawiono wykres wartości funkcji Rastrigina 2D gdzie współczynnik mutacji wynosi 0.2 a całkowita ilość cykli 10000. Aby zwiększyć czytelność wykresu użyto skali logarytmicznej.

Na wykresie uwzględniono również różne maksymalne wielkości populacji, co zaznaczono innym kolorem.

Wykres na rysunku 4.7 pokazuje, że w początkowych cyklach ewolucji populacji gdy osobnicy nie są jeszcze dobrze rozwinięci kolejne rozwiązania znajdywane są wolniej. Szybkość znajdywania nowych rozwiązań przyśpiesza wraz ze wzrostem ilości cykli oraz wzrostem wielkości populacji.

Na rysunku ... przedstawiono wykres wartości funkcji dla większego współczynnika mutacji wynoszącego 0.4.

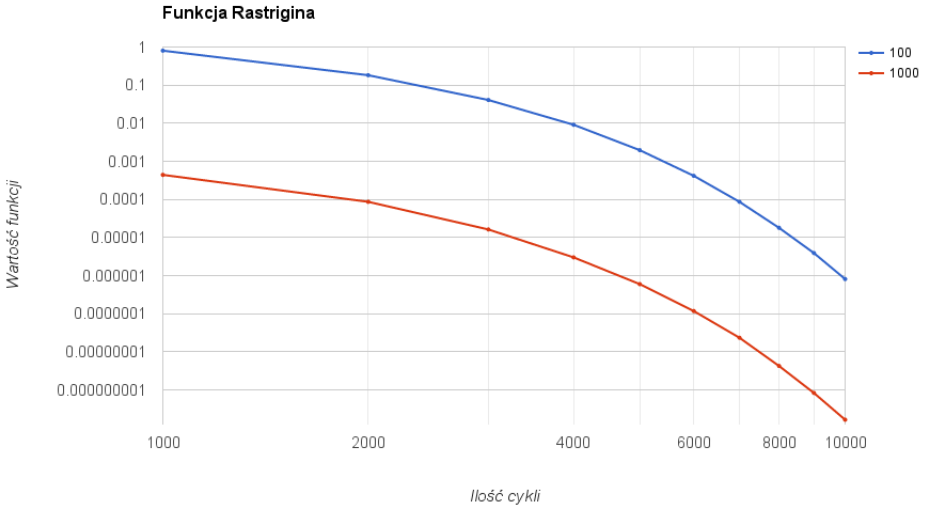
TODO sprobowac zmiescic rozne wspolczynniki mutacji na jednym wykresie, z roznymi odcieniami danego koloru a drugi wykres zrobic dla 8 wymiarow.

Powtorzyc dla 0.4.

Powtorzyc dla 0.8.

Podsumowując, zauważono, że większy współczynnik mutacji przyśpiesza znajdywanie nowych rozwiązań.

Warto dodać, że podczas przeprowadzania testów w klastrze pracowały dwa typy urządzeń Raspberry Pi, model B oraz B plus, który jest nowszy od modelu B oraz posiada większą ilość zaistalowanej pamięci RAM [18].



Rysunek 4.7: Całkowity czas wykonania ... Znaleziony punkt ..., wartość funkcji ...

Rysunek 4.8: Całkowity czas wykonania ... Znaleziony punkt ..., wartość funkcji ...

Dostępna w urządzeniach pamięć RAM ma dość duży wpływ na czas obliczeń. Czas uzyskiwany przez nowszy model urządzenia, który posiada więcej pamięci był około 30% (**sprawdzić ile dokładnie**) lepszy.

4.2.2. Skalowalność

Kolejnymi dwoma parametrami konfigurującymi zadanie są współczynnik migracji oraz częstość jej występowania.

Mają one bezpośredni wpływ na skalowalność platformy. Współczynnik migracji podawany jest w zakresie 0..100 i oznacza procent migrującej populacji. Natomiast parametr określający częstość występowania migracji wyznacza co ile cykli następuje migracja. Jego wartość może wynosić zero, wtedy migracja nie występuje lub jakiś ułamek maksymalnej liczby cykli.

Bez występowania migracji osobników pomiędzy wyspami (węzłami klastra) mogłaby zaistnieć sytuacja, gdy pomimo pracy kilku urządzeń w klastrze każde z nich uzyskałoby podobne wyniki w zbliżonym czasie, co oznaczałoby, że równie dobrze obliczenia mogłyby być przeprowadzane tylko na jednym urządzeniu, bo wzrost liczby urządzeń nie wpłynął znacząco na szybkość znajdywanych wyników.

Dzięki zastosowaniu migracji, gdy do klastra zostaje włączone nowe urządzenie i zaczyna wykonywać obliczenia, to po odebraniu osobników migrujących z innej wyspy, którzy z dużym prawdopodobieństwem będą zawierali lepszej jakości rozwiązania aniżeli nowo zainicjalizowani osobnicy na aktualnej wyspie ewolucja takiej wyspy zostaje przyśpieszona. Przy wysokiej częstości migracji oraz dużym procencie migrującej populacji istnieje ryzyko, że wyspy zostaną zdominowane przez osobników o podobnych cechach, co może prowadzić do odnalezienia jedynie lokalnego minimum zamiast minimum globalnego.

Tutaj chciałbym pokazać jaki jest wpływ migracji oraz jej parametrów na wyniki obliczeń porównując wyniki z poprzedniego podrozdziału uzyskane dla jednego urządzenia..

Wykres ... Na wykresie przedstawiono innym kolorem różną częstość migracji dla 10% migrującej populacji.

4 urzadzenia, 2 wymiary.

4 urzadzenia 8 wymiarow.
Powtorzyc dla 20% migrujacej populacji.
40%.

4.2.3. Wydajność

Średnie zużycie pamięci RAM oraz obciążenie procesora w trakcie wykonywania obliczeń (w zależności od różnych ustawień początkowych) oraz w stanie bezczynności.

4.2.4. Narzut komunikacji.

Częstość wykonywania zapisów do bazy, przesyłania wyników częściowych do klienta oraz wpływ wielkości populacji.

Pozostałym parametrem konfigurującym zadanie jest częstość przesyłania wyników częściowych do klienta co ma wpływ również na częstość wykonywania zapisów do bazy.

4.2.5. Koszty rozbudowy

Krótka analiza kosztów rozbudowy klastra, biorąc pod uwagę ceny urządzeń SoC oraz faktyczny przyrost wydajności uzyskany przy zakupie kolejnego urządzenia.

5. Podsumowanie

W poniższym rozdziale zostało zawarte podsumowanie niniejszej pracy. Opisano również wnioski oraz nakreślono możliwości dalszego rozwoju.

5.1. Wnioski

W ramach niniejszej pracy udało się spełnić początkowe założenia i przygotować platformę oferującą możliwość uruchomienia obliczeń na wielu urządzeniach pracujących w klastrze, dostarczającą narzędzi do monitoringu, rozwiązującą problem load-balancingu zadań, skalowalności a także zapewniającą wsparcie w sytuacji awarii tychże urządzeń, poprzez reorganizację oraz odzyskiwanie sprawności czyli powrót do stanu klastra sprzed wystąpienia awarii. Platforma pozwala również na prowadzenie efektywnych obliczeń mimo ograniczonej mocy obliczeniowej urządzeń SoC. Wykorzystanie tych urządzeń oraz technologii działających na maszynie wirtualnej Javy zapewnia mobilność tego rozwiązania.

Technologie Akka okazały się kluczowe z punktu widzenia niniejszej pracy idealnie wkomponowując się w wybraną tematykę. Dzięki rozwiązanion wysokiego poziomu ułatwiły one tworzenie nowych komponentów oraz dostarczyły pewnych strategii i wzorców zwiększając tym samym przejrzystość jak i możliwości dalszego rozwóju przygotowanego rozwiązania.

Niniejsza praca pozwoliła również autorowi na zapoznanie się z tematyką problemów występujących w klastrze oraz rozwiązaniami stosowanymi w celu zrównoleglenia lub rozproszenia obliczeń. Kolejnym ciekawym aspektem niniejszej pracy z punktu widzenia autora, który pozwolił na eksplorację nowych obszarów wiedzy jest względnie nowa dziedzina w programowaniu a mianowicie programowanie reaktywne i technologie z platformy Typesafe w połączeniu z urządzeniami SoC oraz Cluster Computing.

5.2. Możliwości rozwoju

Rozwiązanie stworzone na potrzeby niniejszej pracy posiada kilka ciekawych aspektów nad którymi badania mogą być kontynuowane. Kilka z nich zostało opisanych poniżej.

5.2.1. Wiele klientów

W testowanym rozwiązaniu rozważono tylko jedną aplikację kliencką, będącą jedynym źródłem zadań w klastrze. W kolejnych etapach prac nad stworzoną platformą warto byłoby się pokusić o zaimplementowanie przypadku gdzie klientów jest więcej i w różnym stopniu obciążają oni urządzenia pracujące w klastrze a także nad sposobem dysponowania wolnymi węzłami tak aby jeden klient nie zajął wszystkich dostępnych węzłów podczas gdy reszta nie byłaby w żaden sposób obsługiwana.

5.2.2. Sieć rozległa

Na potrzeby niniejszej pracy rozważono działanie urządzeń klastra w sieci lokalnej lecz przygotowane rozwiązanie mogłoby się również sprawdzić w sieci globalnej Internet, gdzie węzły klastra mogłyby być uruchomione na różnego typu urządzeniach znacząco oddalonych od siebie i mających

do siebie dostęp za pomocą protokołu TCP/IP. Ciekawym aspektem tutaj byłaby zróżnicowana przepustowość połączenia, ponieważ w takim przypadku trzeba byłoby dobrać odpowiednie ustawienia detekcji awarii i arbitralne wybranie dozwolonego czasu opóźnień mogłoby być niełatwym zadaniem.

5.2.3. Wykorzystanie GPU

Do obliczeń wykonywanych w obrębie platformy wykorzystano jedynie procesory CPU lecz większość urządzeń SoC posiada również jednostki GPU, które są w tym czasie niewykorzystywane. Jedno z urządzeń SoC, zwane Parallela posiada nawet mocniejszy procesor graficzny aniżeli jednostkę centralną. Warto byłoby się zastanowić nad potencjalnym wykorzystaniem tych jednostek np. przy użyciu takich bibliotek jak ScalaCL [29], które pozwalają na wykonywanie kodu napisanego w Scali na procesorach graficznych i równoległym prowadzeniu obliczeń na różnych architekturach. W roku 2011 Martin Odersky, współtwórca języka Scala oraz jego zespół pracujący na uniwersytecie w Lozannie uzyskali wielomilionowy grant od Europejskiej Rady ds. Badań Naukowych (ERBN) dzięki któremu mogli kontynuować pracę nad problemami zrównoleglania obliczeń w różnych modelach programowania jak np. OpenMP, MPI, CUDA, OpenCL [40] co potwierdza, że prace nad podobnymi rozwiązaniami są prowadzone już od wielu lat.

5.2.4. Inne algorytmy

Kolejną kwestią którą można rozważyć w planach dalszego rozwoju jest zaimplementowanie innych algorytmów, które będą w stanie wykorzystać potencjał stworzonej platformy. Warto byłoby również poeksperymentować z różnymi modyfikacjami algorytmów genetycznych, takimi jak różne metody selekcji lub krzyżowania, aby uzyskać jeszcze większą efektywność obliczeń.

5.2.5. Optymalizacja

Urządzenia SoC na których testowano przygotowaną platformę posiadają ograniczone możliwości obliczeniowe, co oznacza, że uruchamiane na nich aplikacje powinny być zoptymalizowane pod tym kątem. Wykorzystanie pamięci RAM oraz procesora jest tutaj bardzo istotne i może w znaczny sposób wpłynąć na szybkość obliczeń. Nie można zatem pozwolić sobie na zbędne operacje i nadmierną zajętość pamięci, co mogłoby wydłużyć wykonywanie obliczeń. Warto byłoby użyć jakiegoś narzędzia do profilowania JVM aby sprawdzić czy nie ma żadnych wycieków pamięci a także przenalizować zrzut pamięci (tzw. head dump) pod kątem instancjonowanych obiektów oraz zajętości pamięci. Można byłoby również wykonać tuning GC (garbage collector) oraz spróbować dobrać optymalne parametry.

5.2.6. IoT

W ostatnich latach coraz bardziej wzrasta znaczenie urządzeń podłączonych do Internetu. Coraz więcej sprzętu codziennego użytku posiada zainstalowane układy procesorowe o wystarczającej mocy obliczeniowej aby móc obsłużyć systemy mobilne takie jak Android. Stwarza to zatem pole do popisu oraz nowe pomysły wykorzystania stworzonego rozwiązania w wersji bardziej rozproszonej.

Bibliografia

[1] Actor references, paths and addresses.

[2] Actor systems.

[3] Akka cluster specification.

[4] Akka cluster usage.

[5] Akka microkernel.

[6] Akka persistence.

[7] Akka persistence mongo.

[8] Akka remoting.

[9] Algorytm genetyczny.

[10] D3.js.

[11] Distributed publish subscribe in cluster.

[12] Dokumentacja angularjs.

[13] Dokumentacja bazy mongodb.

[14] Dokumentacja frameworka akka dla języka scala.

[15] Dokumentacja języka scala.

[16] Dokumentacja play framework dla języka scala.

[17] Dokumentacja projektu akka cluster singleton.

[18] Dokumentacja raspberry pi.

[19] Event sourcing.

[20] Event sourcing pattern.

[21] Funckja rastrigina.

[22] Master/slave.

[23] Play akka cluster sample.

[24] Protokół gossip.

[25] Protokół websocket - rfc6455.

[26] Reactive manifesto.

[27] Round-robin.

[28] Sbt native packager plugin.

[29] Scalacl.

[30] Sigar.

[31] Soc.

[32] Template akka distributed workers.

[33] Type safety.

[34] Work pulling pattern.

[35] An international journal of computing and informatics. *Informatica*, 23, 1999.

[36] T. Alexandre. *Scala for Java Developers*. Packt Publishing, 2014.

[37] J. Armstrong. *Programming Erlang: Software for a Concurrent World (Pragmatic Programmers)*. The Pragmatic Bookshelf, 2013.

[38] K. B. An introduction to cluster computing using mobile nodes. *Emerging Trends in Engineering and Technology (ICETET)*, 2009.

[39] D. M. Bruce Eckel. *Atomic Scala*. Mindview LLC, Crested Butte, CO, 2013.

[40] Z. M. A. R. T. S. A. H. P. O. M. O. K. Chafi, Hassan; DeVito. Language virtualization for heterogeneous parallel computing, 2010.

[41] S. J. Cox. Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing*, Czerwiec 2014.

[42] M. Dudek. Badanie efektywności wielopopulacyjnego algorytmu ewolucyjnego. Akademia GĂłrniczo-Hutnicza w Krakowie, 2009.

[43] A. Freeman. *Pro AngularJS*. Apress, 2014.

[44] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Publishing Co., Styczeń 1989.

[45] M. K. Gupta. *Akka Essentials*. Packt Publishing, 2012.

[46] D. S. H. MĂźhlenbein and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.

[47] C. R. Lin. Adaptive clustering for mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 2006.

[48] C. B. M. Z. Marc Shapiro, Nuno PreguiÂ¸ca. A comprehensive study of convergent and commutative replicated data types. *INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE*, 2011.

[49] L. S. Martin Odersky and B. Venners. *Programming in Scala, Second Edition*. Artima, Grudzień 2010.

[50] M. A. M. Mohamed. Cluster computing with mobile nodes: A case study. *Distributed and Object Systems Lab, Department of CS & E, Indian Institute of Technology Madras, Chennai, India*.

[51] P. R. Nicolas. *Scala for Machine Learning*. Packt Publishing, Grudzień 2014.

[52] R.B.Patel and M. Singh. Cluster computing: A mobile code approach. *Department of Computer Engineering, M.M. Engineering College, Mullana-133203, Haryana, India*, 2006.

[53] J. Richard-Foy. *Play Framework Essentials*. Packt Publishing, 2014.

[54] D. WHITLEY. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[55] Wikipedia. Rss.

[56] L. Wolf. Droidcluster: Towards smartphone cluster computing - the streets are paved with potential computer clusters. *Technische Universitat Braunschweig, Institute of Operating Systems and Computer Networks*.

Dodatek A

.1. Instrukcja uruchomienia platformy

.1.1. System

.1.2. Wymagana infrastruktura

.1.3. Skrypty

.1.4. Aplikacja

Moduły

Api, model danych

Backend

Frontend

Boot

Konfiguracja

Uruchomienie